AUTOMATIC DETERMINATION OF RECOMMENDED TEST COMBINATIONS FOR ADA COMPILERS

THESIS

James Stuart Marr
Captain, USAF

AFIT/GCS/ENG/90D-09

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY
AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

(1)

AFIT/GCS/ENG/90D-09

AUTOMATIC DETERMINATION OF RECOMMENDED TEST COMBINATIONS FOR ADA COMPILERS

THESIS

James Stuart Marr
Captain, USAF

AFIT/GCS/ENG/90D-09

AFIT/GCS/ENG/90D-09

AUTOMATIC DETERMINATION OF RECOMMENDED TEST COMBINATIONS FOR ADA COMPILERS

THESIS

Presented to the Faculty of the School of Engineering
of the Air Force Institute of Technology
Air University
In Partial Fulfillment of the
Requirements for the Degree of
Master of Science (Computer Science)

James Stuart Marr, B.S.
Captain, USAF

December, 1990

Preface

The purpose of this research was to investigate techniques for automatic identification of recommended test combinations for Ada compilers. From the outset, this task was already considered "far more intuitive" than any "reasonable algorithm" could be expected to handle. With the contractual development of such a tool deemed impractical, I possessed a research topic with a wide-open challenge. While the prototype program I developed is still dependent on a certain amount of "human intuition", it did demonstrate the potential benefit of using such automated techniques for identifying recommended test combinations for Ada compilers.

I am grateful to several individuals for their contributions toward the completion of this thesis. To begin my background research, I solicited information from the computer network community. Several individuals were kind enough to E-Mail references to published works or experts in the field of compiler testing. To each of them I extend my thanks. In particular, I must express my appreciation to Glenn Kasten of Ready Systems, California for providing the Gen test case generator that played a key role in this whole effort. Without it, I would not have been able to develop my prototype. I also thank Steve Wilson of the ACVC Maintenance Organization and Deborah Rennels of New York University for providing important background information on this problem and the advances being made in related applications. I thank my thesis advisor, Maj Pat Lawlis, for her guidance and comprehensive review of several iterations of thesis drafts. I also thank my committee members, Maj Dave Umphress and Maj Jim Howatt, for their help in introducing me to this research topic and for their assistance as readers.

Above all, I thank my Lord Jesus Christ for giving me the ability and endurance to reach this culmination of my AFIT experience. And finally, I wish to thank my wife Melissa for supporting me throughout this endeavor.

Commit thy way unto the Lord, trust also in Him, and He shall bring it to pass.
Psalm 37:5

James Stuart Marr

Table of Contents

Page

List of Figures

List of Tables

List of Acronyms

ACEC – Ada Compiler Evaluation Capability

ACVC – Ada Compiler Validation Capability

AFIS – Ada Features Identification System

AFIT – Air Force Institute of Technology

AI – Ada Issue

AIG – Ada Compiler Validation Implementers' Guide

ALIANT – Ada Language Index Analyzer Tool

AMO – ACVC Maintenance Organization

ANSI – American National Standards Institute

ASCII – American Standard Code for Information Interchange

AVF – Ada Validation Facility

BNF – Backus-Naur Form

BSD – Berkeley Software Distribution

BSI – British Standards Institution

CAMP – Common Ada Missile Packages

COBOL – Common Business Oriented Language

DB_{acvc} – Database of Ada features tested by ACVC

DB_{poss} – Database of possible combinations of Ada features

DB_{sw} – Database of Ada features used in DoD software

DoD – Department of Defense

FCCTS – Federal COBOL Compiler Testing Service

ISO – International Standards Organization

L_{ai} – Listing of ACVC tests affected by AI

LG – Linear Graph

LGN – Linear Graph Notation

LHS – Left Hand Sides

L_{nunt} – Listing of unused and untested Ada features

L_{red} – Listing of redundant ACVC tests

LRM – Language Reference Manual

L_{unt} – Listing of used but untested Ada features

L_{want}	Listing of Ada features users want tested
NYU	New York University
PAT	Program Analyzer Tool
PPG	Pascal Program Generator
RADC	Rome Air Development Center
RHS	Right Hand Sides
SEMANOL	A formal notation for language specification
TGG	Text Generator Generator
VSR	Validation Summary Report
YACC	Yet Another Compiler Compiler

AFIT/GCS/ENG/90D-09

Abstract

Ada compilers are validated using the Ada Compiler Validation Capability (ACVC) test suite, containing over 4000 individual test programs. Each test program focuses, to the extent possible, on a single language feature. Despite the advantages of this "atomic testing" methodology, it is often the unexpected interactions between language features that result in compilation problems. This research investigated techniques to automatically identify recommended combinations of Ada language features for compiler testing. A prototype program was developed to analyze the Ada language grammar specification and generate a list of recommended combinations of features to be tested. The output from this program will be used within the Ada Features Identification System (AFIS), a configuration management tool for the ACVC test suite. AFIS is being developed by the ACVC Maintenance Organization (AMO). The prototype uses an annotated Ada language grammar to drive a test case generator. The generated combinations of Ada features are analyzed to select the combinations to be tested. While the skill and intuition of the compiler tester are essential to the annotation of the Ada grammar, the prototype demonstrated that automated support tools can be used to identify recommended combinations for Ada compiler testing. (KR)

AUTOMATIC DETERMINATION OF RECOMMENDED TEST COMBINATIONS FOR ADA COMPILERS

I. Introduction

Functional testing is a commonly used technique for validating programming language compilers. It "...is the process of executing a series of generally independent tests designed to exercise the various functional features of a software product " (32:1051). Each test case is usually designed to evaluate a limited number of language features. This practice simplifies the testing process by focusing on the specific objective of the test, while minimizing interactions between language features. Unfortunately, it is often the unexpected interactions between language features that result in compilation problems. Although it may be possible to develop a test case for each language feature, it is impractical to develop tests for all combinations of features. Therefore, compiler test suite developers must determine which combinations of language features to test.

This research investigated techniques to automatically identify recommended combinations of Ada language features for compiler testing. A program was developed to analyze the Ada language grammar specification and generate a list of recommended combinations of features to be tested. The output from this program will be used within the Ada Features Identification System (AFIS), a configuration management tool for the Ada test suite. AFIS is being developed by the Ada Compiler Validation Capability (ACVC) Maintenance Organization (AMO).

1.1 Ada Background

The development of the Ada programming language originated in the Common High Order Language Program, a Department of Defense (DoD) sponsored activity that began in 1975 (39:11). Among several factors leading to the creation of a standard programming language, and associated environments, were the need to

- Reduce the Cost of Developing Systems.
- Increase the Portability of Software.
- Increase the Portability of Software Developers.
- Increase Productivity.
- Increase Reliability and Maintainability.
- Support the Management of Complexity and Change. (39:3)

The Ada requirements and design process involved several thousand contributors including "...more than 50 people [who] were intimately involved in some facet of the design" (12:13). The culmination of the "Ada effort" was the completion of the Ada Programming Language Reference Manual, ANSI/MIL-STD-1815A, in 1983 (16). To make sure that all Ada implementations would conform to this standard, the DoD began researching validation technology and procedures long before any Ada compilers were available.

> It is to the government's credit that Ada [is] the first programming language to have [had a] means for enforcing the specification as well as an analysis of potential implementation difficulties and oversights available when they [would] do the most good—before too great an investment [was] made in diverse (and probably divergent) implementation efforts and (even more important) before a large user population [came] to depend on nonconforming compilers. (21:58)

1.2 Ada Compiler Validation Capability

"In September, 1979, SofTech, Inc. started work on the Ada Compiler Validation Capability (ACVC), an effort aimed at developing conformity tests for Ada compilers" (23:195). The ACVC is the means by which an Ada compiler is tested to insure compliance with the requirements of the Ada Programming Language. The ACVC consists "...of the test suite, the support programs, the ACVC user's guide and the template for the validation summary report" (1:3). A validated and certified Ada compiler implementation is one that has successfully passed the ACVC tests according to the procedures outlined in (1). The ACVC Maintenance Organization (AMO) at Wright-Patterson AFB, Ohio, provides the technical and administrative support required to produce and distribute ACVC versions, and perform quality control and configuration management on the ACVC test suite.

1.3 Ada Features Identification System

The Ada Language is continually subject to new interpretations and refinements that affect the ACVC test suite. As *Ada Issues* (AIs) are distributed, the AMO must identify affected ACVC tests for modification manually. "With 4000 tests in the test suite and a large and growing number of AIs, hand identification is becoming increasing ineffective" (3:1). For example, one recent revision to the ACVC validation suite contained "...more than 400 changes compared with the previous set of tests, ...[A]bout 400 of those changes were just clarifications and about 43 were substantive in nature" (7). These and other challenges led the AMO, in early 1988, to propose the development of the Ada Features Identification System (AFIS). The AFIS would be used to identify which Ada language features need to be tested in combination, identify redundant ACVC tests, identify, for modification, those ACVC tests affected by AIs, and provide a smooth transition of the ACVC to the next Ada standard, Ada 9X (44, 5, 6).

The proposed AFIS consists of three parts: the Ada Language Index Analyzer Tool (ALIANT), the Program Analyzer Tool (PAT), and the associated database management system. The ALIANT would identify the combinations of dependent features that exist in the Ada language and output the information to the database (DB_{poss}). The PAT would identify those combinations used in operational DoD software and in the current version of the ACVC, and output the information to the AFIS database (DB_{sw} and DB_{acvc}). The AFIS database would be queried to obtain the following types of information: a list of all tests affected by an AI (L_{ai}), a list of all redundant ACVC tests (L_{red}), a list of unused and untested combinations (L_{nunt}), and a list of used but not tested combinations (L_{unt}). The L_{nunt} listing would be distributed to Ada users to "...determine which of these combinations they would like to use but can't due to current compiler limitations (L_{want})" (3:para c.5.8). The L_{want} listing would be used to decide what types of tests should be added to the ACVC test suite. Figure 1.1 summarizes the AFIS requirements and Figure 1.2 is a diagram of AFIS.

1.4 Problem Statement

The goal of this research was to develop an ALIANT prototype that would automatically identify recommended combinations of Ada features for compiler testing. The original AFIS statement of work was issued in 1988 (3). New York University (NYU) is currently under contract with the AMO to implement the PAT and the AFIS database. They chose not to attempt implementation of the ALIANT portion of the AFIS due to

RUN TOOLS	PRODUCT
ALIANT	DBposs (Database of possible combinations)
PAT against ACVC	DBacvc (Database of tested combinations)
PAT against DoD s/w	DBsw (Database of currently used combinations)
QUERY	
DBacvc for features identified in AIs	Lai (List of all tests affected by AIs)
DBacvc for multiply tested features	Lred (List of redundant tests)
DBposs and not DBsw and not DBacvc	Lnunt (List of unused and untested combinations)
DBposs and DBsw and not DBacvc	Lunt (List of used but not tested combinations)
SEND	
Lnunt to users	Lwant (List of unused combinations users wanted)

Figure 1.1. (44)

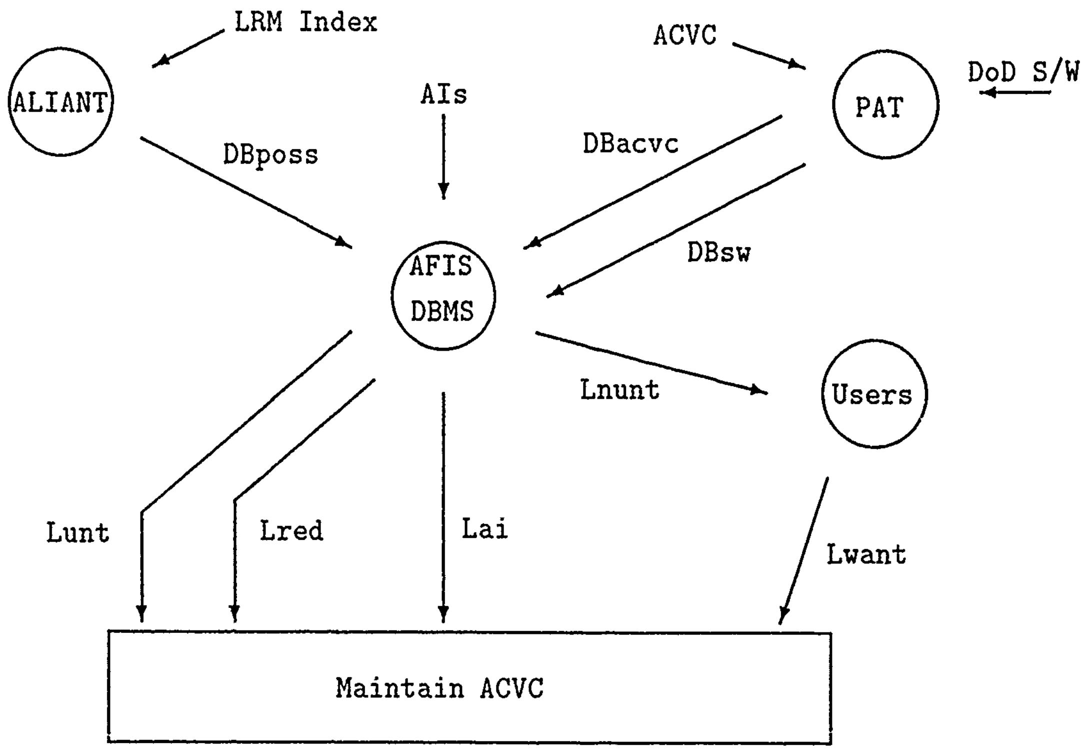

Figure 1.2. (44)

the combinatorial complexity of determining the combinations of Ada language features to test (34).

> With regard to compilers it is certainly true that it is not practical to test all possible combinations of language components and data types. ...[but] it is certainly possible to test all reasonable combinations. What is "reasonable" is admittedly a subjective judgement, but such subjectivity regarding test limits is hardly unique to software testing. (32:1051)

The problem is how to identify the "reasonable" combinations that should be tested. As the ALIANT name implies, the original statement of work for AFIS suggested analysis of the Ada Language Reference Manual (LRM) index to determine the recommended combinations of Ada features to test. A more formal definition of the relationships among Ada language features exists in the Ada grammar as found in Appendix E of the LRM (16).

> Upon first examination, it was believed that feature dependencies were revealed by extracting each of the features and nested subfeatures from the index. Unfortunately, identifying dependencies proved to involve far more intuitive judgment than at first believed. No reasonable algorithm has been developed to effectively extract combinations of dependent features from the index. (35:3)

The AFIS is clearly incomplete without the ALIANT capability. To reap the full advantage of the AFIS capability, a method must be developed to generate the recommended combinations of Ada language features.

1.5 Scope

The scope of this research effort was limited to the identification of the recommended combinations of Ada features. This research did not attempt to generate the test cases containing the recommended combinations. The focus was on demonstrating the feasibility of the proposed ALIANT subsystem of AFIS.

The type of feature combinations generated by this research attempted to parallel the *primary features* identified for the PAT subsystem. The PAT development team identified a set of 297 primary features from the Ada grammar and from terms in the Ada LRM index.

> In most cases, the features are either nonterminals of the syntax summary, or major terms of the index (those printed in boldface), or both. In some cases, one feature is a general or basic term, and a few other features are special cases of that general feature. For example, the feature "generic_formal_type" has subcases "generic_formal_type:discrete_type", "generic_formal_type:integer_type", etc. (35:8)

These primary features have effectively taken simple combinations of features and given them a name so they can be treated as atomic features. Within the PAT subsystem, these primary features are used to identify the features that exist in particular test patterns used to search the ACVC test suit or DoD software. A detailed description of how PAT uses these test patterns and primary features can be found in (35).

The recommended combinations of Ada features generated by the ALIANT prototype do not specify the order or context of the features. The PAT subsystem must be used to determine if a specific permutation of a given *primary feature* combination occurs in the ACVC test suit or DoD software.

1.6 Development Environment

The availability of the following resources was assumed during the development of the solution design:

- The Gen compiler test case generator (See Appendix A for a description of the Gen software).
- An Ada programming environment which would allow interface to the Gen software which is written in *C*.
- An electronic copy of the Ada grammar which would be annotated to meet the input requirements of the Gen software.

The Verdix Ada programming environment, hosted on the AFIT Galaxy computer (Elxsi-6400 with 4.3BSD UNIX operating system), was chosen as the development environment for the ALIANT prototype. That computer system provided the necessary Ada support tools and was compatible with the requirements of the Gen software. The electronic copy of the Ada grammar was obtained through the Ada Information Clearinghouse services to simplify the entry of the annotated grammar input for the ALIANT prototype.

1.7 Approach

The purpose of this research was to investigate techniques for automatic identification of recommended Ada compiler test combinations. These techniques have been applied to the developn · nt of an ALIANT prototype, which uses the Ada language grammar as input.

The first step in the research was the completion of a survey of literature related to this topic. Next, the Gen program was used to investigate generation of Ada test cases. Initial tests used a subset of the Ada grammar, modified to meet the particular input requirements of the Gen software. Since the generator tool can potentially generate an infinite number of combinations, experimentation with the Gen randomness constructs was required to limit the combinations to a "reasonable" number. After refining the techniques on a subset of Ada, the test case generation was extended to include the full Ada grammar. Next, a program was developed to analyze the Gen output. This program was used to search the generated Ada language combinations and perform a tabulation of the most frequent combinations of two features, three features, and so on. An interface between the analyzer program and a prototype AFIS database was developed to complete the ALIANT prototype. The final step of this thesis was analyzing the feasibility of implementing the ALIANT validation tool, based on the ALIANT prototype. The analysis includes recommendations for further study and improvements.

1.8 Thesis Overview

Chapter II documents the results of the literature review and provides background material for this thesis. The emphasis is on Ada compiler validation and methods for generating compiler test cases.

Chapter III outlines the design of the problem solution. This design describes the steps required to implement the problem solution and indicates what resources/tools are used to solve the problem.

Chapter IV covers the implementation of the problem solution. It describes any difficulties encountered during implementation and discusses what decisions were made, if any, to modify the solution design.

Chapter V includes the final analysis of the problem solution. The results of the solution (ie. generated computer products) are presented and their meaning is discussed.

Finally, Chapter VI contains conclusions about the overall thesis effort and recommendations for further study. Supplementary material and computer generated products are included as appendices to this thesis.

II. Literature Survey

This chapter provides background material related to this research effort. It begins with a brief discussion of compiler testing in general and then describes the Ada Compiler Validation Capability (ACVC). The ACVC development approach, validation procedures and ACVC limitations are presented. Next, several formal language specification techniques are described including attribute grammars, denotional semantics and high-level semantics. Finally, four different research papers concerning automated compiler testing are summarized. Some of the techniques used in these papers will be applied to this thesis research.

2.1 Compiler Validation and Testing

"Software validation ... [is] the process of testing a completed software product in its operational environment" (32:1051). Compiler validation checks the conformance of a compiler implementation to the applicable language standa. ' and differs from the validation of applications software in several respects:

- Validation systems must be capable of functioning on a variety of dissimilar hardware and operating systems.
- The staff performing the validation is not involved in the development or the maintenance of the products being tested.
- The results of a validation could impact the eligibility of the product for procurement. (32:1051) (10)

The most commonly used method for testing compilers is *functional testing.* As mentioned in Chapter I, "functional testing is the process of executing a series of generally independent tests designed to exercise the various functional features of a software product" (32:1051). This testing method is considered to be the "most thorough technique presently available" for testing software. Since compilers have a formal specification, the grammar, they are especially "amenable" to construction of functional tests for each feature. In fact, a later section in this chapter will discuss tools that will automatically generate compiler test cases from the grammar that defines the language. Even with such automated tools, exhaustive testing is impractical, if not impossible. Therefore, compiler validators must select a reduced set of test cases that will achieve nearly the same confidence level as exhaustive testing. A technique applicable to limited combinations of independent language features is *orthogonal Latin squares.* According to Robert Mandl, this technique " ...yields the informational equivalent of exhaustive testing at a fraction of the cost" (31:1054).

Most validation test suites for standard languages still rely on time consuming manual generation of test cases. For example, the Federal COBOL Compiler Testing Service (FCCTS) and the Ada Compiler Validation Capability (ACVC) manually develop individual test cases for compiler testing. In most cases, this is satisfactory because once the test suite is developed it remains static except for occasional changes to the language standard that require the test suite to be updated. Later sections of this chapter will discuss how automatic generation may be used as a complement to such manually prepared test suites (32) (23) (14) (15).

The compiler validation process only determines the conformance of a given compiler to the associated language standard. A validated compiler is one that meets the requirements of each test case in the validation suite. Validation is not the same as evaluation, in which a compiler is tested for such factors as efficiency and speed. A validated compiler may not be suitable for certain applications for various reasons, such as memory limitations or machine dependencies. Additional evaluation techniques are used to determine the fitness of one compiler or another for a specific user application (40) (20) (27) (13).

2.2 Ada Compiler Validation Capability (ACVC)

The ACVC test suite is used to validate Ada compiler implementations. This section examines the background of the ACVC and describes the procedures used to validate a compiler.

2.2.1 ACVC Development Approach. The development of the ACVC test suite began before the Ada language Standard was published. "The decision to establish an independent test team before Ada's design was even near completion was essential to the success of the ACVC effort and helped considerably in improving the precision of the eventual Standard" (23:211). From the start, the policy was established that would require a compiler implementation to pass "all applicable correct tests" to be usable on DoD projects (23:201).

The development philosophy for the ACVC test suite included the requirement for many small test cases. Each test case was designed to test a limited number of Ada features to minimize the impact of a failed test and to simplify identification of the feature that failed the test. Particular attention was given to those parts of the language that are hard to implement. This philosophy is intended to make certain all compiler developers implement the language carefully and completely. The ACVC test programs were developed manually at an average cost of "8 person-hours" for each test (23:211).

> The [ACVC] test suite is updated continually and released periodically. Updates are needed to correct errors in tests. In addition, new tests are added, and sometimes existing tests are strengthened. Sometimes tests have to be changed because of interpretations recommended by the Language Maintenance Committee/Panel. (23:208)

The ACVC test suite consists of over 4000 test files in the following six classes:

- Class A : legal Ada programs that should compile successfully.
- Class B : illegal Ada programs that should not compile successfully.
- Class C : legal Ada programs that should compile and execute successfully.
- Class D : tests that check compiler capacity limits.
- Class E : executable tests that check implementation dependent options.
- Class L : illegal Ada programs that should be detected at link time. (2:1-4,1-5)

This classification of test cases shows the "breadth of test coverage and helps automate the analysis of test results" (23:60). Although there are a large number of individual test cases, many of the tests can be chained together to reduce the amount of manual intervention required to validate a compiler. Much of the analysis of test results is automated to improve responsiveness and reliability. Examples of each class of test cases are provided in Appendix B.

The Ada standard permits some features to vary among compiler implementations. Some ACVC test cases are designed to determine the behavior of a compiler with regard to such characteristics as nesting of loops, expression evaluation, rounding methods, input and output features, and so on. The results of such test cases are reported for informational purposes in a Validation Summary Report (VSR). Such features as maximum length of an input line or the maximum precision in floating-point *type declarations* will differ between compilers. Therefore, some of the ACVC test cases are "templates" that include test parameters to make the test case compatible with the compiler being validated. After providing appropriate values for these parameters and adding any required job control statements, the test cases are submitted to the compiler for testing (21:62).

2.2.2 Ada Compiler Validation Implementers' Guide. The Ada Compiler Validation Implementers' Guide (AIG) (22) was developed in response to the DoD's requirement that the ACVC contractor produce "a report to aid compiler developers":

> The report should identify common errors in Ada compilers, describe compiler implementation techniques that will avoid difficulties, and provide exemplary programs that illustrate potential trouble spots in conforming to the standard and that clarify the intended interpretations of the standard. (21:58)

The AIG is written and used in parallel with the Ada Language Reference Manual (LRM). Each section in the AIG corresponds to a section in the LRM that describes a particular feature of the language. The AIG section contains up to seven subsections as follows:

- Semantic Ramifications – documents semantic implications that might not otherwise be obvious from a reading of the LRM.
- Legality Rules – explicitly lists context-sensitive syntactic and semantic legality rules to be checked by an Ada translator prior to beginning execution of an Ada program.
- Exception Conditions – explicitly lists the conditions under which an implementation is required to raise an exception associated with some predefined language construct.

- Test Objectives and Design Guidelines – specifies the validation tests to be written, lists the problems to keep in mind while writing test cases under "Implementation Guidelines", and, when necessary, outlines the program structure required to satisfy a test objective.
- Approved Interpretations – summarizes approved interpretations of the LRM which correct errors, ambiguities, or inconsistencies.
- Changes from July 1982 – describes changes to the draft LRM, dated July 1982, that affect the Ada feature describe in this AIG section.
- Changes from July 1980 – describes changes between the July 1980 and July 1982 LRM drafts. (22:1-1, 1-2)

2.2.3 Validation Procedures. The ACVC test suite is released for a six month review period before it is used in validation tests. During that time, compiler implementers or other parties may submit comments to the ACVC Maintenance Office (AMO). "At the end of six months, the new version of the ACVC is released for validation use for a period of 18 months" (1:9). The procedures for validating an Ada compiler are specified in the Ada Compiler Validation Procedures (1). The *validation by testing* is accomplished in the following six steps:

- Validation Agreement – the compiler implementer becomes a *customer* of an Ada Validation Facility (AVF) by formal agreement.
- Prevalidation – the customer tests the candidate Ada compiler using a customized ACVC test suite and submits results to the AVF.
- Validation Testing – the AVF tests the candidate compiler using the customized ACVC test suite and compares with the prevalidation tests.
- Declaration of Conformance – the customer declares the availability of a validated Ada compiler.
- Validation Summary Report – a Validation Summary Report (VSR) is produced by the AVF describing the extent to which an Ada compiler conforms to the Ada standard.
- Validation Certificate – a Validation Certificate is issued that expires one year after the expiration date of the ACVC version used for the validation. (1) (2)

Under certain conditions, an Ada compiler implementation may be *validated by registration.* This method is used in cases where a validated compiler is changed for "corrective, adaptive, or perfective" reasons within the "scope of software maintenance" (1:17). These procedures allow minor modifications to be made to a validated compiler without having to reaccomplish the entire *validation by testing* process.

2.2.4 ACVC Limitations. The ACVC validation suite determines the conformance of a compiler implementation with the Ada standard, but it does not give any indication of its quality. As was mentioned earlier, *validation* is not the same as *evaluation.* To select an Ada compiler for a particular application, the user must consider other requirements such as speed, memory availability, support tools, etc. "Although over 200 validated Ada compilers are available for more than 25 computer architectures, compiler technology has been inadequate to support many of Ada's features" (19:59). The Ada Compiler Evaluation Capability (ACEC) is a test suite designed to evaluate the performance characteristics of Ada compilers, a task the ACVC was never intended to handle (40) (41).

An example of known limitations of current ACVC test suites concerns the use of *generic units.* During the development of the Common Ada Missile Packages (CAMP) software by McDonnell Douglas Astronautics Company in St. Louis, contractors noted that "validated Ada compilers frequently cannot handle any but the simplest generic units" (24:75). The CAMP contractors pointed to the fact that most ACVC test cases are designed to test a single objective. As a result, some of the more complicated cases are not tested and validated compilers may not be able to support a "complex mix of generic units, essential to the use of dynamic reusable software" (24:75). They recommended that more complicated test cases be added to the ACVC test suite to remove the inadequacies noted during the CAMP project (24) (37) (18).

The configuration management of over 4000 ACVC test cases is becoming a formidable task. Some test cases are redundant and periodic updates of test cases continually change the composition of the ACVC test suite. Currently, the identification of test redundancies and other maintenance functions are done by hand. The ACVC Maintenance Office (AMO) has undertaken the development of several automated tools that will improve the configuration management capability of the ACVC. As described in the introduction, the purpose of this research was to investigate the feasibility of one of those automated tools, the Ada Language Index Analyzer Tool (ALIANT). The capability to automatically identify test combinations for Ada compilers will help improve the ability of the ACVC to validate Ada compiler correctness (3).

2.3 Automatic Compiler Testing

Advances in automated techniques for compiler testing are closely related to the methods investigated in this research for identifying recommended test combinations for Ada compilers. This section presents the background on developments in annotated grammars and compiler test case generators. Similar techniques were applied to the development of the ALIANT prototype.

2.3.1 Annotating Grammars. The first step in developing automated compiler testing tools is to determine the format of the input grammar that will guide the generation of compiler test cases (See Appendix A.1, for a brief summary of grammar notation or

(33) for more in depth coverage of grammar and compiler terminology). A context-free grammar can generate programs that are valid syntactically but invalid semantically. In other words, some valid programs can have invalid meaning. A language grammar can also generate an infinite number of programs or infinitely long programs; therefore, additional grammar constructs are required to limit test case generation to meaningful and reasonably-sized programs. The following discussions present some research efforts in the area of adding semantics to grammars.

2.3.1.1 SEMANOL specification. A 1978 Rome Air Development Center (RADC) research effort (9) investigated methods for automated compiler test case generation. This research was spawned by the apparent inadequacy of existing compiler test suites. The RADC researchers noted that existing test suite development methods were

- Not systematic.
- Not designed with reference to measures of test effectiveness.
- Prepared manually.
- Expensive. (9:2)

The SEMANOL specification language, developed as part an earlier RADC research effort, is used to define a language's grammar and the associated context-sensitive features (this RADC report did not give the origination of the SEMANOL acronym). "A formal SEMANOL specification of a programming language is a program; a program for processing a source language program text written in the programming language being defined" (9:5). The SEMANOL *metalanguage*, or language that describes another language, is combined with the context-free grammar of a computer language to form the specification. The specification consists of declarations, control commands, context-free syntax, and semantic definitions (9:10). Although a considerable degree of human intervention is still required to set up the SEMANOL specification, great benefits in reduced test case generation time and increased test quality and consistency can be realized. This research effort established a design framework for further experimentation and implementation (9).

2.3.1.2 Attribute grammars. Attribute grammars are used for the formal specification of the semantics of a programming language. The development and use of an attribute grammar for Ada is described in (17). Formally, an attribute grammar consists of:

- a *context-free grammar.*
- a set of *attributes* for each symbol of the context-free grammar.
- *attribution rules* establishing the value of every attribute according to the syntactic production in which it appears and in terms of the values of other attributes of symbols in the same production.
- *conditions* involving attributes of one production. If a given condition is not satisfied by the attribute values of a particular subtree, a specific error message is given. (17:9)

An example of attributes in the Ada language is its strong *type checking.* The context-free grammar may define a statement to allow two identifiers separated by an operator such as "+". The syntax is valid for any two identifier names formed by a legal combination of characters. However, if one identifier was declared to be an integer type and the other is a character type, the semantics of the language require an error to be generated. Compilers must include the appropriate routines to check attributes of various language symbols. Attribute grammars are a tool to specify such semantics within the context-free grammar.

Such attribute grammars are used in test case generation tools to insure the creation of meaningful test cases in terms of syntax and semantics.

2.3.1.3 Denotional and High-Level Semantics. Denotional semantics is a formal method for giving mathematical meaning to programming languages. "Originally used as an analysis tool, denotional semantics has grown in use as a tool for language design and implementation" (36:xi). As described in previous sections, the syntax of a programming language can be described quite well with a context-free grammar. The semantics or meaning of the sentences or programs generated by a language are more difficult to represent formally. Denotional semantics is a method that is used to specify, in formal notation, the meaning of a program. "The denotional semantics method maps a program directly to its meaning, called its denotation. The denotation is usually a mathematical value, such as a number or a function" (36:3). These formal specifications are being used in several research efforts that "...demonstrate the possibility of automatically generating compilers for no[illegible] languages from formal semantic descriptions" (29:3). These same principles could be applied to automatic generation of the test cases to test a compiler.

Unfortunately, the compilers generated from such "classical" formal specifications tend to have performance characteristics much worse than handwritten compilers (29:3). As a result, a new style of semantic description is being developed called *high-level semantics.* The "...high-level descriptions are easier to write and comprehend than traditional denotational specifications ...[and] realistic compilers can be straightforwardly generated from high-level descriptions" (29:4). As mentioned before, lessons learned in compiler generation can be applied to test case generation as well.

2.3.2 Compiler Test Case Generators. Several research efforts have investigated the automatic generation of compiler test cases. This section summarizes four research reports describing the development of compiler test case generators.

2.3.2.1 An Automatic Generator for Compiler Testing. This article by Franco Bazzichi and Ippolito Spadafora (8) examines a method for automatically generating compilable test programs. At the time this article was written, several methods had been "...studied and developed to test compilers ...however, none of these methods ...[had] solved the problem completely and efficiently" (8:343). The objectives of this study were twofold. First, "...to automatically generate compilable programs for different programming languages, rapidly and cost-effectively" (8:343). The authors adopted a *context-free parametric grammar* which is a grammar containing additional context-sensitive aspects. They used the grammar as an input to an algorithm that would produce a set of compilable programs. The second objective was to "...to generate incorrect programs in a controlled way, using the above-described methodology" (8:343). The incorrect programs would be used to demonstrate that a compiler would reject a program containing errors.

Features of Complete Compiler Tests. The ideal method of testing a computer program is to run enough test cases so that every path in the code is executed at least once. Unfortunately, the manual preparation of such exhaustive tests is very time consuming. When a compiler is analyzing a source program, there is a clear relationship between the states the compiler traverses and the syntax rules that were used to generate the original source code. "Therefore, it is reasonable to assume that a fairly complete test of a compiler should include a set of programs containing all the syntactical entities of the language: the set of test programs should be derivable from the source grammar, using each [rule] of the grammar at least once" (8:344). But there also must be a "criterion of minimality" to keep the size of generated test programs within certain efficiency limitations. For this reason, Bazzichi and Spadafora chose the criterion of "shortest derivation" when generating test programs (8).

Limitations of Automatic Generation. The automatic generation of test programs has certain limitations. Without knowing the inner workings of the compiler under test (*black box testing*), the automatically generated test cases cannot be expected to detect all the errors a compiler may have. The involvement of experienced compiler implementers is essential to understanding what types of errors are most likely. "The type of errors which implementers may make must be hypothesized and tests must be constructed which can only succeed if there are no errors present" (8:344). Therefore, the authors of this study included a "flexible set of directives" that allow the generated test cases to be produced in a "controlled" way according to the various parameters provided by the compiler implementers (8).

2.3.2.2 Independent Testing of Compiler Phases Using a Test Case Generator. This article by William Homer and Richard Schooler (25) considers the problems associated with independent unit testing of compiler phases. The discussion focuses on "...the testing of a large compiler whose modules, or phases, communicate via complex graph-structured intermediate representations" (25:1). The design of any large computer program in modules with well defined interfaces allows the development of each part to progress independently of one another. Each module can be unit tested before final integration of all components. "The obvious source of unit test inputs for a particular phase would be the preceding phases, but staffing, scheduling, and technical considerations often lead to parallel, or 'out of order', development of the phases" (25:2). As was mentioned in the previous article, the manual fabrication of test cases is too time intensive for anything beyond a few simple test inputs. The authors of this article present a generator that takes as input a context-free grammar with some context-sensitive constructs to insure meaningful test cases will be generated.

Linear Graph (LG) System. The compiler tested in this study uses the LG (Linear Graph) System. LG provides compiler development tools and uses a human-readable notation called LGN (Linear Graph Notation). The LGN "...was meant to facilitate creation of test inputs and examinations of phase outputs" (25:3). The LGN format can be used to define test inputs to later compiler stages without having the earlier stages completed. The LG system will convert the LGN into the proper internal data structures for execution. All that is needed is a way to automatically generate the large LGN tests that would be very difficult to produce manually. The authors used a tool called Text Generator Generator (TGG) to automatically generate the LGN test cases. TGG inputs a context-free grammar and generates strings of the grammar. As with the previous study, this test case generator allows special "context-sensitive" constructs to be added to the grammar to control the types of strings generated and to provide "pseudo-random" behavior (25).

2.3.2.3 Automatic Generation of Executable Programs to test a Pascal Compiler. This article by Dr. C. J. Burgess (11) describes work done to develop a means to automatically generate compiler test programs that are immediately compiled and executed by the compiler under test. This differs from the methods discussed in the previous two articles, in that those methods only generated the source code for test programs. The programs still had to be individually compiled and tested. This article specifically concentrates on the generation of test programs for Pascal compilers. The programs are generated from an attribute grammar and "...contain self-checking code so that only those programs that fail to execute correctly will produce any output. Thus, by examining the output file, those test programs that have failed to run correctly can be quickly identified, even though a large number of programs may have been run" (11:304). Dr. Burgess feels that the combination of automatically generated tests and manually generated tests should increase a tester's confidence that the compiler is correct.

Testing a Compiler. The test case generator system described by Dr. Burgess is designed to produce a specified number of test programs, compile and execute each one, and produce error messages for any tests that failed. If a test program compiles and executes without errors, the only output produced for that test program will be a Pascal comment listing the values used for random number seeds and the number of the test program. If, however, the test program produces a compilation error, or fails at execution time, additional error messages will be output. The source code for each test program is discarded after execution, and the next test program is generated. This process continues until the desired number of test programs have been generated, compiled, and executed. The output listing can then be examined for error messages. If error messages are found, "...the comment just before the output will contain the seeds from which the source of the test program which failed can be recreated. This means that a large number of programs can be generated and run through the compiler with very little extra demand on file space" (11:315).

2.3.2.4 Experience with a Compiler Testing Tool. This report by B. A. Wichmann and M. Davies (42) describes a compiler testing tool called the Pascal Program Generator (PPG). As with the Ada language, the Pascal programming language has a standard validation suite that is used to validate Pascal compiler implementations.

> The [validation] service, offered by BSI [(British Standards Institution)] and its licencees [sic] world-wide, ensures that validated compilers conform closely to the ISO-Pascal Standard. While the 700 or so test programs in the validation suite ensure that the syntax and semantics of the language are supported adequately by validated compilers, it is nevertheless the case that validated compilers still have significant bugs. (42:3)

The "bugs" that cause the most concern are those "...in which the compiler generates incorrect code from a legal program" (42:3). The PPG is being developed as a complementary testing tool to the existing Pascal validation suite. Given a few inputs for a random number generator, it will generate self-checking executable Pascal programs.

Testing Experience. The PPG is still under development. Many issues must be resolved, but the initial indications are that such an automated tool could be very useful to compiler developers. The PPG can generate very large and complex tests that may fail due to a bug that is unlikely to occur in normal use. For this reason, it is too early to suggest incorporating the use of such generators in formal validation suites. However, compiler-writers could benefit from such test generators for "in-house" development testing. These tools may help identify errors that would not be detected by the validation suite. The authors of PPG are considering a similar implementation for the Ada programming language. They expect that such an implementation will be more difficult to accomplish since Ada is more complex than Pascal (42:13).

2.4 Conclusion

Since grammars define the valid sentences in a given language, they can be used to automatically generate compiler test cases. The four research articles presented in this chapter are examples of the progress that has been made toward developing automated compiler testing tools . Most mature standard languages already have a comprehensive test suite that has been manually developed over several years. Automated tools could be used to supplement the formal validation process for these languages. In particular, the Ada validation process may benefit from these compiler testing tools. The automated testing tools are perhaps most valuable during compiler development/research in which a compiler-writer needs the ability to quickly generate many test programs. Further research in this area is sure to yield new techniques for determining the correctness of programming language compilers.

The Gen compiler test case generator, similar to the automated tools described in this chapter, was used in this research. Appendix A describes Gen in detail and the next chapter explains how Gen was used to develop an ALIANT prototype.

III. Solution Design

This chapter describes the solution design chosen to address the problem of determining recommended Ada compiler test combinations. As appropriate, background on other design alternatives is provided to show what options were considered.

3.1 Objective

The overall objective of this problem solution was to develop a prototype of the ALIANT subsystem of AFIS, the proposed ACVC configuration management tool. The ALIANT prototype was designed to input an annotated Ada BNF grammar, analyze possible combinations of Ada features, and output the recommended combinations of Ada features. A context diagram of the ALIANT prototype is shown in Figure 3.1. As indicated in the context diagram, the prototype consists of two parts: the Gen software and the analysis/selection procedures. The Gen software will generate feature combinations according to the annotated Ada grammar (see Appendix A), while the remaining procedures will analyze and select recommended combinations. Figure 3.2 contains the process descriptions for the ALIANT prototype design presented in this chapter.

3.2 Annotating the Ada Grammar

The first design consideration was deciding how the Ada grammar would be annotated to meet the input requirements of Gen, the first phase of the ALIANT prototype. The grammar annotation is a manual process that must be completed before the ALIANT prototype is executed. Before the various Gen randomness constructs could be added to

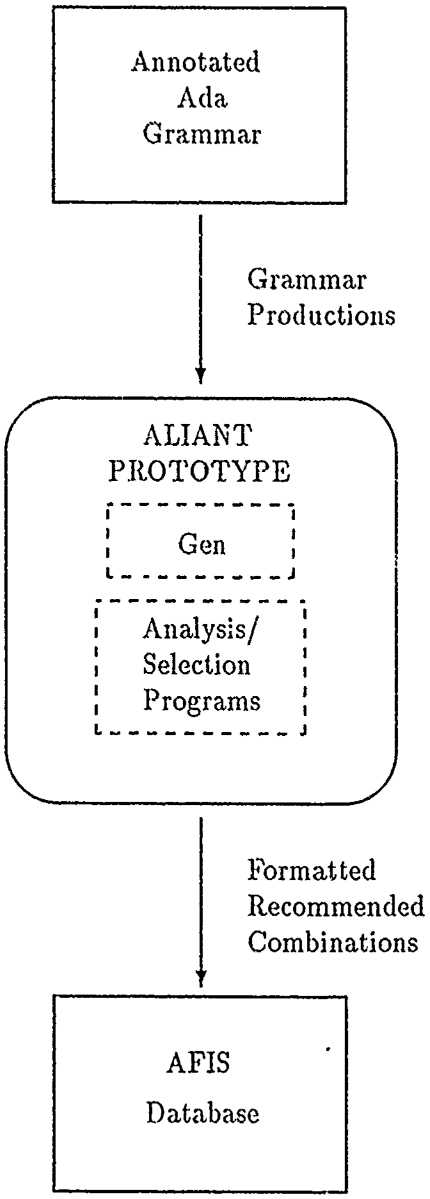

Figure 3.1. ALIANT Prototype Context Diagram

1.0 Generate Combinations.

Input An annotated Ada grammar in Gen compatible format.

Process This process is the Gen software. Test cases are generated according to the annotated grammar. The number of combinations generated is determined by the *generation statement*, which is the last entry in the annotated grammar.

Output Test case combinations containing character strings that were enclosed in quotes in the annotated grammar.

2.0 Identify Combinations.

Input Test case combinations from Gen.

Process Combinations are uniquely identified according to the collection of features they contain. Null combinations and duplicate combinations are discarded. Information stored about each combination will include the number of features in the combination, the number of times each feature occurred in the combination, and the number of duplicates of the combination.

Output A data store containing the above information for each identified combination.

3.0 Recommend Combinations.

Input Data store containing identified combinations.

Process Combinations are selected from all identified combinations according to the number of times a combination was duplicated and the number of features a combination contains. These threshold values are entered by the user at runtime.

Output Data store containing recommended combinations.

4.0 Load AFIS Database.

Input Data store containing recommended combinations.

Process The recommended combinations are stored in the AFIS database format. This process translates the data store format into the database bit-matrix format.

Output Recommended combinations in AFIS database format.

Figure 3.2. ALIANT Process Descriptions

the Ada grammar, it had to be in a form that Gen would recognize. For example, Figure 3.3 shows the Ada grammar productions for the *identifier* as it appears in Appendix E of the Ada LRM. The direct translation of the grammar productions in Figure 3.3 to the Gen input format is shown in Figure 3.4. Note the addition of the productions to generate digits and letters and the ident_tail production to handle the *zero or more* option in the original identifier production.

```
identifier ::= letter {[underline] letter_or_digit}

letter_or_digit ::= letter | digit

letter ::= upper_case_letter | lower_case_letter
```

Figure 3.3. Ada Identifier Grammar Rules

```
identifier = letter ident_tail

ident_tail = (("_" | "") letter_or_digit ident_tail) | ""

letter_or_digit = letter | digit

letter = upper_case_letter | lower_case_letter

upper_case_letter = [A-Z]

lower_case_letter = [a-z]

digit = [0-9]
```

Figure 3.4. Gen Input for Identifier Grammar Rules

Gen is designed to generate valid, compilable test cases for a given grammar. For this research, it was necessary to "remember" what language features were used to generate a particular test case. For example, when the Gen compatible grammar of Figure 3.4

is executed by Gen, a series of randomly constructed identifiers is produced. Although this is what Gen was designed to do, the identification of the Ada features that were used to generate the identifiers is lost (i.e., ident_tail, letter_or_digit, letter, digit). Since the purpose of the ALIANT prototype is to recommend combinations of features, the actual generation of, say, identifiers is not desired. With this objective in mind, a different approach was considered. By disregarding anything but the *nonterminals* in each Ada grammar rule, the input to Gen can be modified so that the output only contains the names of the Ada features used to generate the test case. Whenever a nonterminal is to be expanded by a production rule, Gen can be instructed to output the literal string representing that nonterminal. Referring back to the identifier example, Figure 3.5 shows how this modified approach would be used. The literal strings in quotes are included in the Gen output.

```
identifier = letter ident_tail

ident_tail = (("underline" | "") letter_or_digit ident_tail) | ""

letter_or_digit = letter | "digit"

letter = "upper_case_letter" | "lower_case_letter"
```

Figure 3.5. Modified Gen Input for Identifier Grammar Rules

Additional *pruning* of the grammar can be accomplished by raising the level at which the literal strings are output. For example, in Figure 3.5, the string "letter" could have been output instead of either "upper_case_letter" or "lower_case_letter". If the lower level details are unnecessary, this choice would simplify the rules and the amount of output produced. Most likely, the annotated grammar would probably go no lower than the level of "identifier" in the example rules presented. Since the existing ACVC test cases handle the simple checks for proper identifier construction, it would only be necessary to note where an identifier appears in a more complex test case. The ALIANT prototype does not need such details as what characters formed the identifier or how long the identifier was.

Based on the options just discussed, the *modified* annotated grammar of Figure 3.5 was chosen to eliminate some of the lowest level detail and to simplify the later analysis requirements of the Gen output. After the original Ada grammar has been manually translated into the Gen format described above, the randomness constructs are added to those rules containing alternatives. The randomness constructs allow the user to tell Gen which alternatives should be chosen more often than others. A detailed description of these constructs is provided in Appendix A. The values chosen for these randomness constructs were based on experimentation and estimates of the most frequently used Ada features. The next chapter will describe the implementation of the grammar annotation process in more detail.

3.3 Processing the Grammar

The first step in the ALIANT prototype design is the generation of possible combinations using the Gen software. This step is represented by process *1.0 Generate Combinations* in Figure 3.6. Initially, a partially annotated subset of Ada was used to determine the optimum techniques for generating the test cases. Then, the lessons learned were applied to larger and larger subsets of Ada until the complete Ada language was included. The details of how the Ada grammar subsets were developed is provided in the next chapter. The ALIANT design uses an ASCII data store between Gen and the analysis programs.

This ASCII data store provides a standard text format for storing the feature combinations produced by Gen. This type of interface allows the Gen software to run to completion without requiring complicated synchronization with the Ada analysis programs.

3.4 Analyzing Combinations

The next phase of the ALIANT prototype design consisted of specifying the processes to analyze the Gen output and determine the recommended combinations to test. The purpose of process *2.0 Identify Combinations* in Figure 3.6 is to identify each generated combination and assign a unique identification number to each one. This process discards duplicate combinations and tallies the number of occurrences of each Ada feature for a given test case. Process *3.0 Recommend Combinations* eliminates combinations that include too many language features and chooses the recommended combinations. The output from the analysis phase is in ASCII for later interface to the AFIS database.

3.5 Database Interface

The final step in the solution design was the interface between the ALIANT prototype and the AFIS database. Process *4.0 Load AFIS Database* in Figure 3.6 represents the interface routine that was designed to load the recommended combinations into the AFIS database in the appropriate format compatible with the other AFIS subsystems. For this research, the database format chosen to demonstrate the functionality of this interface was a bit-matrix. For each recommended combination to be stored in the database, a record will be output containing a unique identification number and an array of '0's and '1's. For each possible feature in the combination, a '0' will indicate the corresponding feature is

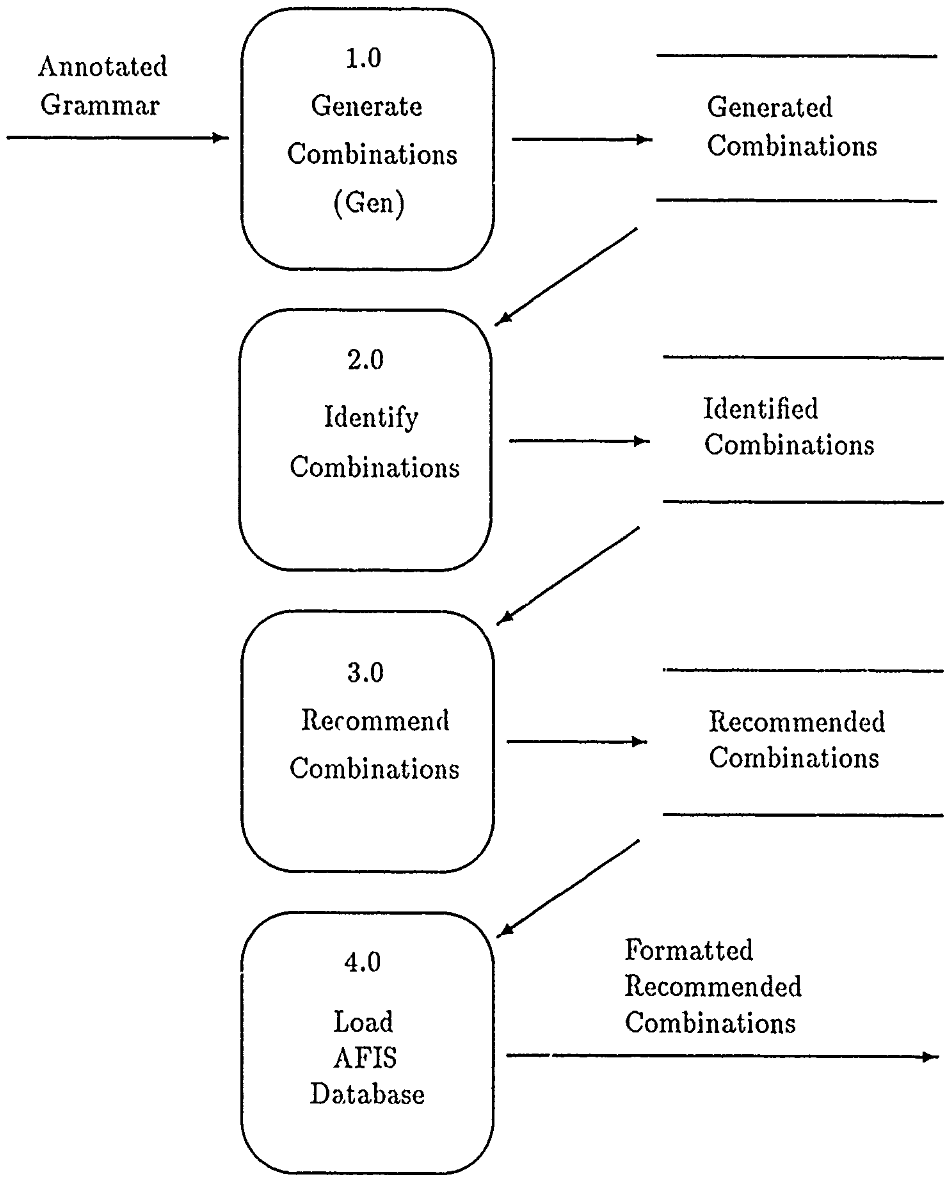
Annotated Grammar
1.0 Generate Combinations (Gen)
Generated Combinations
2.0 Identify Combinations
Identified Combinations
3.0 Recommend Combinations
Recommended Combinations
4.0 Load AFIS Database
Formatted Recommended Combinations

Figure 3.6. ALIANT Prototype Data Flow Diagram

not included in the subject combination. A '1' will indicate the feature is included in the combination. A simple example of the database design is provided below for the case in which there are only twenty features being considered:

```
1:  0 1 1 0 0 1 1 0 0 0 1 1 0 0 0 0 1 0 0 0

2:  1 1 1 0 0 0 0 0 0 1 1 0 0 0 0 0 0 1 0 0

3:  0 0 0 0 0 1 1 1 0 0 0 0 0 0 1 1 1 0 0 0
```

In the example above, the database contains 3 records numbered 1, 2, and 3. Each array of '0's and '1's corresponds to 20 features numbered from left to right starting at 1. The first combination consists of features 2, 3, 6, 7, 11, 12, and 17. Each numbered feature represents an Ada grammar nonterminal symbol. Any program using this database would have to be given the mapping of feature number to the equivalent Ada feature.

The ALIANT design described in this chapter served as a starting point for the prototype implementation. The next chapter will describe the development of the ALIANT prototype using an iterative modeling technique.

IV. Solution Implementation

This chapter describes the details of the solution implementation. First, the annotation of the Ada grammar is described. Next, the interfaces developed between the Gen software and the ALIANT Ada programs are presented, followed by a discussion of the ALIANT prototype development.

4.1 Grammar Annotation

As shown in the previous chapter, the grammar annotation is a manual process. Although some portions could possibly be automated, the only automated aid used in annotating the grammar was that provided by a word processor's global "find and replace" feature. Such a feature was initially used to change all occurrences of "::=" in the Ada grammar to the Gen equivalent "=". Appendix A contains a summary of the Gen input requirements.

Although it was originally thought that a literal subset of the Ada grammar would be used for initial development, it proved to be more difficult to pick out the productions that would be used for one subset, only to add them in later. The solution to this problem was to leave all "left-hand sides" (LHS) of productions intact but to "short-cut" the "right-hand sides" (RHS) by replacing all RHS containing terminal symbols by the corresponding LHS in quotes. This causes Gen to output the names of the LHS if these productions are encountered during test case generation. Productions containing only nonterminals were already in the proper format once the "::=" sign was replaced by the "=" sign, and the Gen randomness operator, "%", was added to each alternative symbol, "|". For RHS that span more than one line, enclosing parentheses are required for Gen. For consistency and clarity, the enclosing parentheses were used for all production's RHS. The following examples show how this was done. The Ada *case_statement* production,

```
case_statement ::=
    case expression  is
        case_statement_alternative
        {case_statement_alternative}
    end case;
```

was replaced by the following statement:

```
case_statement = ( "case_statement" )
```

The Ada *library_unit_body* production,

```
library_unit_body ::= subprogram_body | package_body
```

was simply changed to the following statement:

```
library_unit_body = ( subprogram_body | % package_body )
```

The Gen randomness construct, "%", causes one of the alternatives to be chosen with a probability equal to the other alternatives.

In some cases, additional productions had to be added to model the *zero or more* notation found in the Ada grammar. For example, the highest level production in the Ada grammar, *compilation*, uses the *zero or more* notation as shown below:

```
compilation ::= {compilation_unit}
```

The Gen compatible format of the *compilation* production used in this research is:

```
compilation = ( "START_COMPILATION: "
  ( "" | % compilation_unit more_units ) ":END_COMPILATION \n" )

more_units = ( "" | % compilation_unit more_units )
```

The "START_COMPILATION: " and ":END_COMPILATION \n" strings were added to cause Gen to output clearly marked compilation test cases on separate lines. This feature simplifies visual analysis of the Gen output and also provides a means for the ALIANT prototype analysis programs to identify the start and end of each compilation test case.

The completely annotated Gen compatible grammar, using the method described above, is provided in Appendix E.1. Unfortunately, this first method does not generate detailed test cases. Since most of the productions are simplified on the RHS, the test cases produced are at a very high level. Most of the productions are never reached during test case generation. To demonstrate how the grammar can be modified to provide more detail, another annotated grammar was produced. This second version, shown in Appendix E.2, expands several of the productions that previously were reduced to a single quoted string on the RHS. For example, the Ada production for *generic_instantiation* appears as follows in the Ada grammar:

```
generic_instantiation ::=
    package identifier  is
      new  generic_package_name [generic_actual_part];
  | procedure identifier is
      new  generic_procedure_name [generic_actual_part];
      | function designator is
          new  generic_function_name [generic_actual_part];
```

Since this research was not interested in the actual generation of the nonterminal symbols in the production above, the *primary features* mentioned in Chapter I were used to annotate the grammar. These "pseudo-nonterminals" yielded the following Gen compatible production:

```
generic_instantiation = ( ( "gen_function_instantiation " | %
  "gen_package_instantiation " | % "gen_procedure_instantiation " | %
  "gen_subprog_instantiation " )  ( "" | % generic_actual_part ) )
```

Using the Gen randomness construct, "%", without any numbers means each alternative will be chosen with a probability equal to the other alternatives. An equivalent production using specified probabilities would have the integer 25 following each of the first three percent signs, and the integer 50 following the last percent sign. The production above provides more detail than the previous "short-cut" version:

```
generic_instantiation = ( "generic_instantiation " )
```

The results and analysis of generating test cases using both Gen grammars is discussed in the next chapter.

4.2 Gen to ALIANT Interface

With the annotated grammars completed, the next step in the ALIANT prototype implementation was determining how the output from Gen would be input to the Ada programs for analysis. Since the Gen program is a strictly "batch" operation, the UNIX *redirection* and *piping* capabilities were chosen as the first interface between Gen and the other programs. The UNIX redirection and piping features allow the transfer of data between files and programs without user intervention and are frequently used in script files that are executed in "batch" mode. As the next section will describe in more detail, the analysis part of the ALIANT prototype was initially conceived as a batch operation as well, so the use of redirection and piping was a good choice. The following command line illustrates the use of redirection ("<") to direct an input grammar file (adagen.gen) into Gen (gen.exe), and piping ("|") to send the Gen output to the ALIANT Ada analysis programs:

```
gen.exe < adagen.gen | aliant_driver.exe
```

This method of interfacing the Gen output to the ALIANT analysis programs worked fine in the batch mode of operations. When the evolving prototype of the ALIANT analysis programs led to interactive input requirements, another method had to be employed. The piping of Gen output to the ALIANT driver prevented the ALIANT analysis programs from getting interactive input from the keyboard. All input had to come from the Gen program. The alternative was to output the Gen test cases to an intermediate file and have the ALIANT driver open the file during execution. This method would also allow interactive entry from the keyboard. The second method assumed that the last line of the input grammar file would contain the *generation* statement indicating how many *compilations* to generate (see Appendix A for a review of the Gen input grammar format). Unfortunately, this meant the number of combinations was "hard-coded" within the grammar and had to be manually changed before each execution with a different number of combinations. This inconvenience led to further improvements in the interface implementation.

Through an iterative prototyping process, several detailed requirements were eventually identified for a flexible and *user friendly* input to the ALIANT prototype. Figure 4.1 identifies the key requirements. The method chosen to implement all of these requirements was a UNIX *shell script.* Appendix D.1 shows the commented shell script that implements each user input interface requirement. A dataflow diagram illustrating the operation of the shell script is provided in Figure 4.2. In the diagram, step A must terminate normally before step B will execute. Each rectangle represents a file, keyboard, or screen as indicated. The rounded squares represent processes performed by independent programs or UNIX utilities as detailed in the shell script. A description of the various files mentioned in this diagram and throughout this chapter can be found in Figure 4.3.

4.3 ALIANT Prototype Development

The ALIANT analysis programs were implemented using an iterative *rapid prototyping* approach. Using the high-level design presented in the previous chapter, an initial prototype was developed. From this initial prototype, features were modified and new ones added as detailed implementation decisions were made. Figure 4.4 is an object-oriented diagram of the Ada portion of the ALIANT prototype final implementation. The dependencies between the various Ada packages and driver subprogram are indicated with arrows. For clarity, the dependencies involving two standard library packages, Text_IO and Math, are not shown. Text_IO is used by each compilation unit except Lex_Pkg, while Math is only required by Parameter_Pkg. The corresponding documented source

- The ALIANT prototype will be executed with a single command and up to three command-line parameters.

    1. The root of the filename of the Gen input grammar *without* the generation statement (i.e., this parameter would be *adagen*1 for a filename *adagen*1.*gen*).
    2. The number of compilation combinations to generate. This parameter will be used to create a *generation* statement in the input grammar.
    3. An optional batch input filename that will allow the user to execute an ALIANT session in batch mode with the output going to a default file.

- Error checking will be performed on input parameters and descriptive error messages generated as necessary.
- A Gen *generation* statement for the requested number of combinations will be appended to a copy of the input grammar and submitted to the Gen program.
- The number of requested combinations will be passed, via a file, to the ALIANT Ada programs for creation of required run-time memory space.
- If the Gen program fails to execute normally for any reason, the ALIANT analysis programs will not be allowed to execute.

Figure 4.1. ALIANT Prototype Requirements

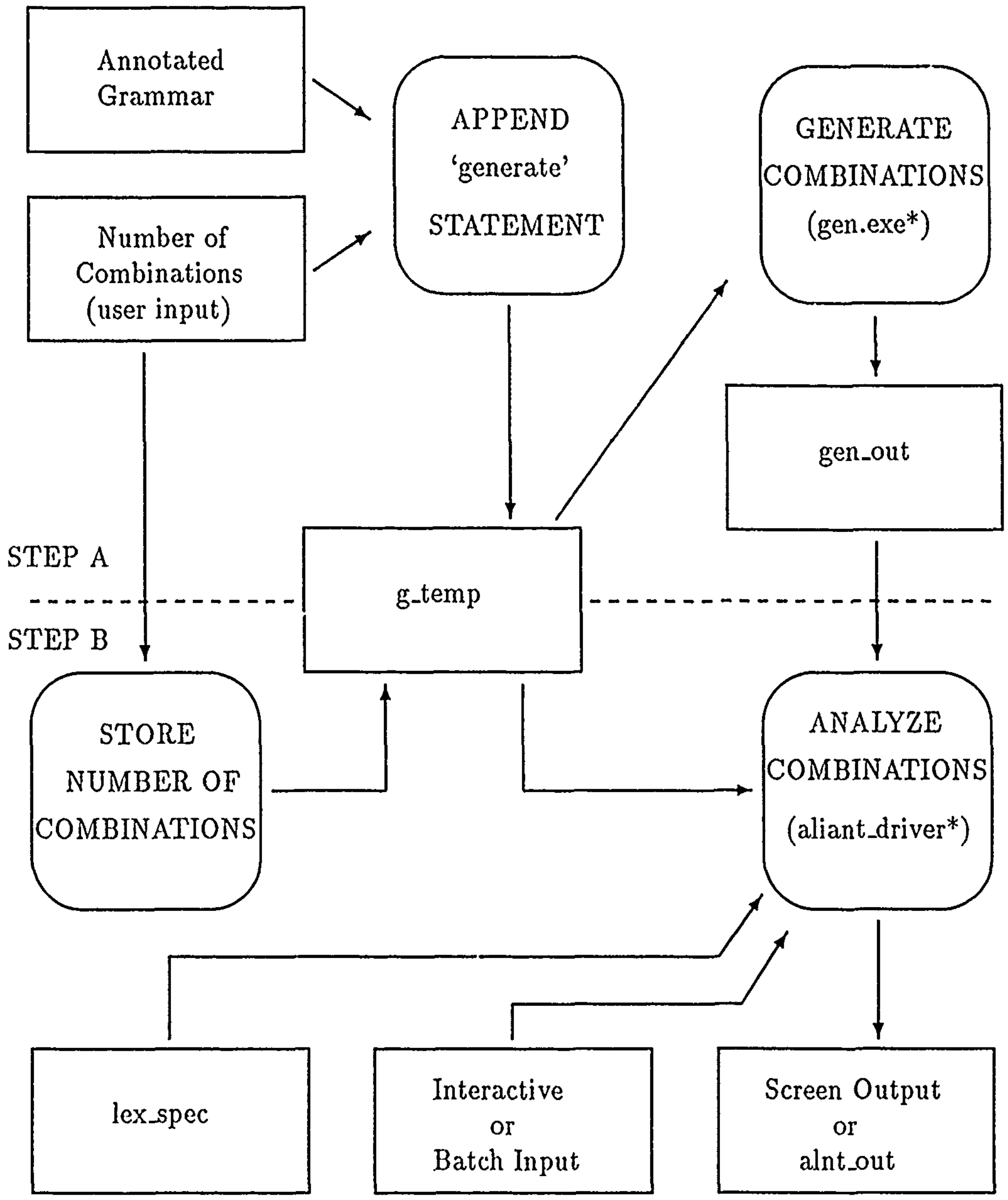
Annotated Grammar
Number of Combinations (user input)
APPEND 'generate' STATEMENT
GENERATE COMBINATIONS (gen.exe*)
gen_out
STEP A
g_temp
STEP B
STORE NUMBER OF COMBINATIONS
ANALYZE COMBINATIONS (aliant_driver*)
lex_spec
Interactive or Batch Input
Screen Output or alnt_out

Figure 4.2. Executing ALIANT Prototype

afis_db Database file produced by ALIANT which contains a record for each selected combination. Each record consists of a unique identification number and a bit array indicating which features are included in the combination (Appendix F.4).

aliant.a ALIANT Ada source code (Appendix D.2).

aliant_driver.exe* Executable ALIANT Ada code.

alnt_out Output file used when ALIANT is executed in batch mode (Appendix F.2).

<filename> User specified filename for batch input to ALIANT. It contains an entry per line corresponding to the user prompts that will be generated for a specific batch execution (Appendix F.2).

gen.exe* Executable C language Gen program.

gen_out File containing output from Gen that is read in by the ALIANT analysis programs (Appendix F.4).

g_temp File used to build the input to Gen containing a copy of <root>.gen appended with the user specified generation statement. After Gen execution is terminated, this file is used to pass the user specified number of combinations to the ALIANT analysis programs (Appendix F.4).

lex_spec Lex specification that is automatically generated by the mk_lspec specified program. The program generated from lex_spec is called as a subroutine (yylex) from the ALIANT analysis driver program (Appendix C.2).

mk_lspec Lex specification and C driver program that is used to create the program that generates the lex_spec Lex specification from the <root>.gen file. The program specified by mk_lspec extracts all Ada nonterminal symbols and pseudo-nonterminals and formats the Lex specification that will identify expected characters and strings output by Gen (Appendix C.3).

<root>.gen Input grammar for Gen (Appendix E).

runa* Executable UNIX shell script used to run ALIANT (Appendix D.1).

yylex File containing the object code of the compiled C-program generated by Lex from lex_spec. This object code is linked with the ALIANT Ada programs using *pragma interface* (Appendix C.1).

Figure 4.3. ALIANT Prototype File Descriptions

code is provided in Appendix D.2. Figure 4.5 provides a tree structure diagram showing the relationships of the various subprograms that make up the ALIANT Ada code. The remainder of this section describes the iterative development of this final version of the ALIANT prototype.

The initial ALIANT prototype was batch oriented with required parameters "hard-coded" in the Parameter_Pkg specification. The first version did not have a Features_Pkg, which will be described later. The Lex_Pkg contains the interface to the *Yylex* lexical analyzer program that returns integer values for each nonterminal or pseudo-nonterminal found in the input file from Gen (see Appendix C for a full description of how Lex is used in this prototype). The first prototype did not use the Opengen and Closegen routines shown in Figure 4.4 since the output from Gen was being "piped" through the standard input file. The Matrix_Pkg provides encapsulation of the storage matrix that is used to tabulate the feature occurrences in each combination. A two-dimensional matrix was chosen for simplicity and speed. In this initial prototype, the maximum size of the matrix was determined before compile time based on the expected number of combinations and features that would be found. Although dynamic storage, using run-time allocated memory, would have provided the most optimal use of memory and allowed growth "without bound", the speed factor was considered most important; and the use of extra memory for the duration of the ALIANT execution was not considered a problem for this data structure. Figure 4.6 contains the descriptions of each of the major routines in the Matrix_Pkg which show how the matrix is used and why high-speed access is desired.

The ALIANT_Driver controls the operation of the ALIANT prototype. The following pseudo code describes the algorithm used to process the feature combinations produced by

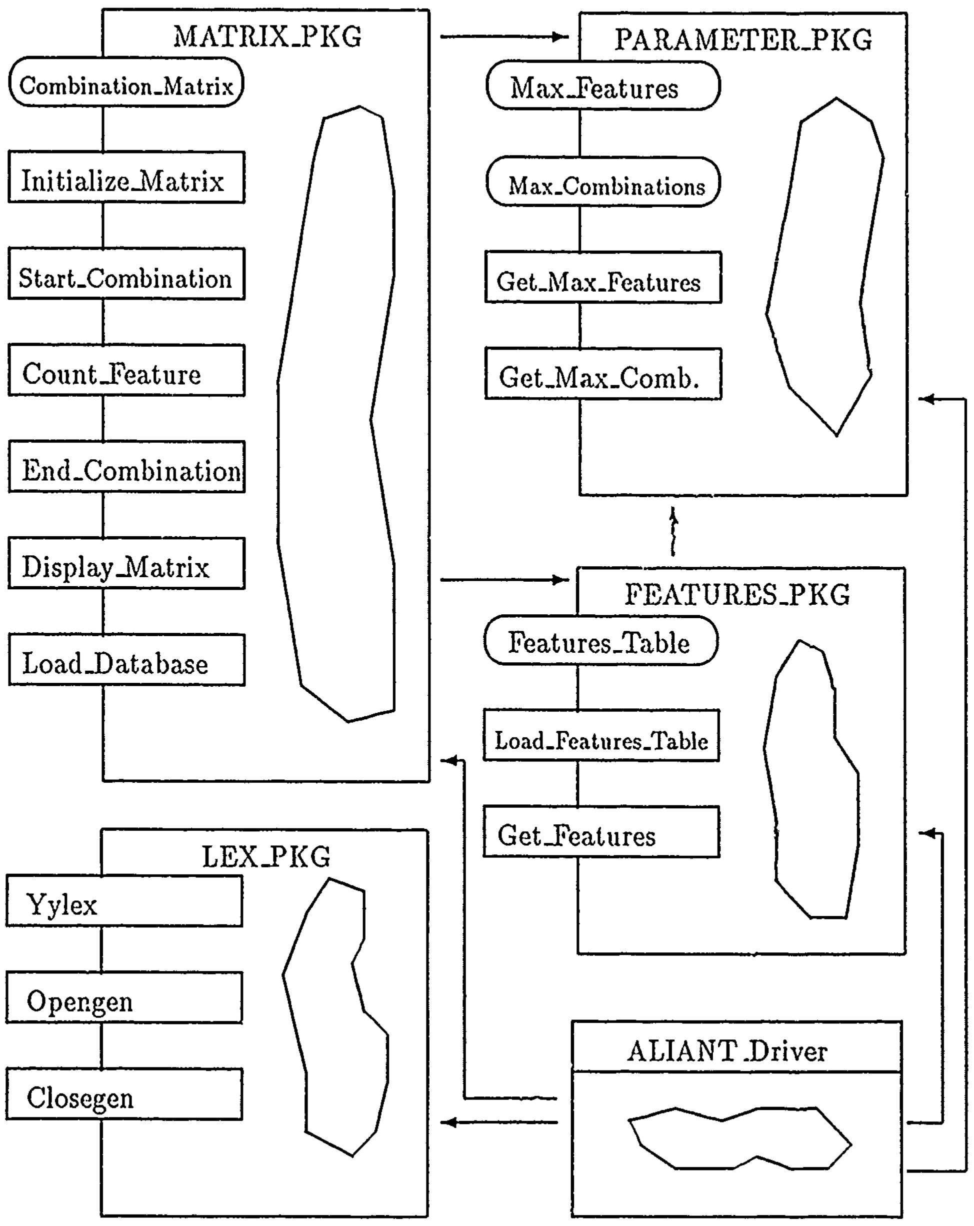

Figure 4.4. ALIANT Object-Oriented Description

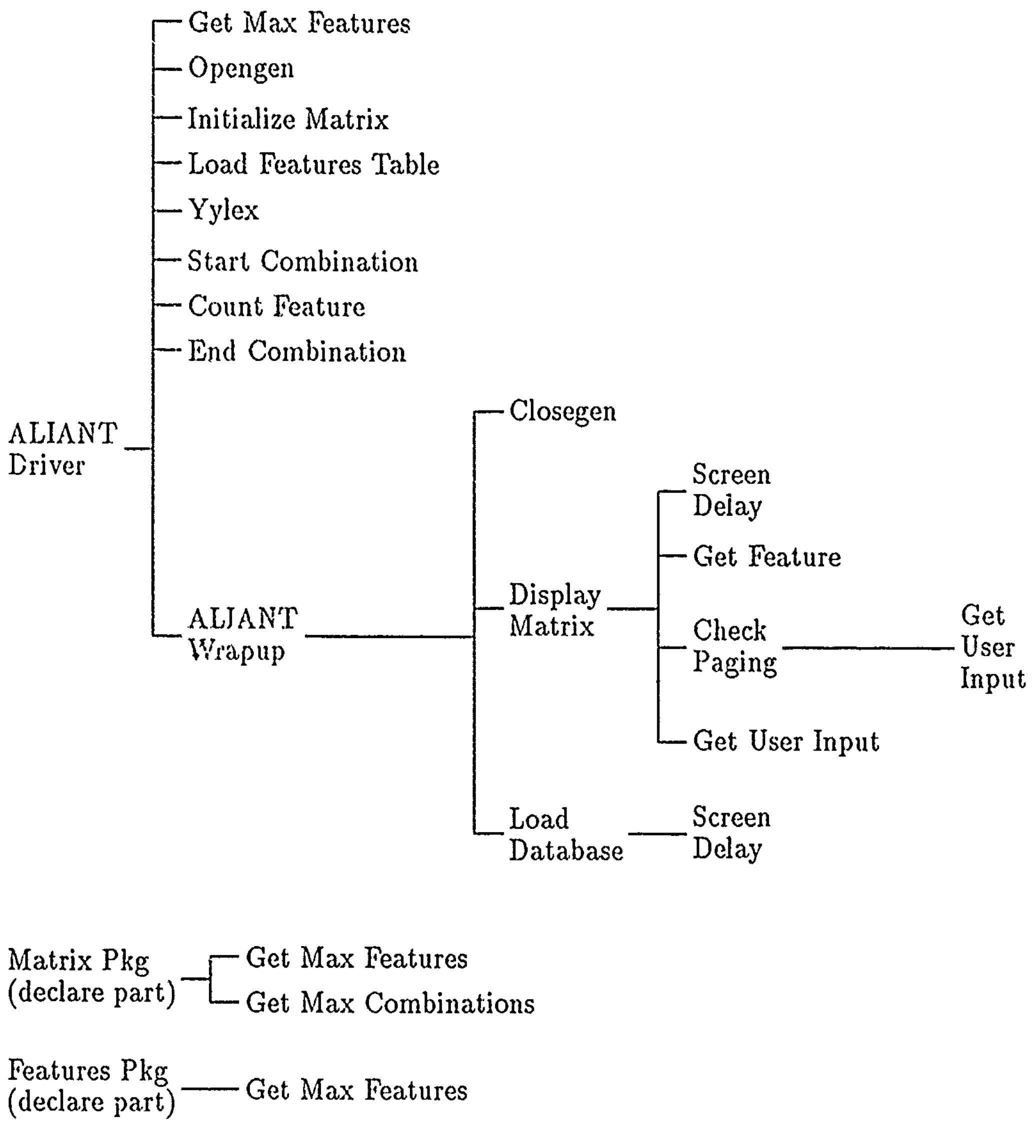

Figure 4.5. ALIANT Structure Diagram

Initialize_Matrix This procedure simply zeros out the combination matrix. There are no parameters.

Start_Combination This procedure increments counters and subscripts to start tabulation for a new combination. There are no parameters.

Count_Feature This procedure increments the feature counter for the current combination being tabulated. An input parameter provides the *feature number* subscript for the combination matrix, while the *current combination* package variable provides the other subscript.

End_Combination This procedure is called when the end of a combination is identified. Tests are performed to determine if this combination is a "null" combination that does not contain any features or is a duplicate of a previous combination. Two combinations are duplicates if both corresponding feature counts are zero or non-zero at the same time. If the current combination is in fact null or duplicate, appropriate counters are incremented or decremented as necessary and the current row of the combination matrix is zeroed out. The tests for duplicate combinations requires comparisons between the current combination and all previous combinations. This is where the speed factor is most important.

Display_Matrix This procedure displays the contents of the combination matrix. The initial prototype simply displayed the count totals for each feature number by combination. Additional features were added as described in later portions of this chapter.

Load_Database This procedure was added in later iterations of the ALIANT prototype. If this option is chosen, ALIANT will output the selected combinations to a file in "bit matrix" format, as described in the previous chapter.

Figure 4.6. Matrix_Pkg Procedure Descriptions

Gen. Note that this pseudo-code includes all features implemented in the final prototype, some which haven't been described yet.

```
begin ALIANT_Driver
    call Matrix_Pkg.Initialize_Matrix
    call Features_Pkg.Load_Features_Table
    call Lex_Pkg.Opengen
    call Lex_Pkg.Yylex (return Token)
    while (not end of Gen input file) loop
        case Token is
            when feature token =>
                call Matrix_Pkg.Count_Feature (send Token)
            when start token =>
                call Matrix_Pkg.Start_Combination
            when end token =>
                call Matrix_Pkg.End_Combination
            when others =>
                print error message
        end case
        call Lex_Pkg.Yylex (return Token)
    end while loop
    call Lex_Pkg.CloseGen
    call Matrix_Pkg.Load_Database
end ALIANT_Driver
```

The initial ALIANT prototype verified that the interfaces between the Gen program, Lex subroutines and Ada analysis programs would work as expected. These preliminary tests revealed a requirement to increase the size of an output buffer in the Gen software to handle some of the larger test cases. Other than this minor adjustment, the Gen software was not modified in any way to meet the requirements of ALIANT.

The initial prototype also revealed that the interface between the Yylex routine and the Ada code required some unexpected modifications to the Yylex source code and the Verdix Ada library. The Yylex routine is created using a utility program called Lex. As described in Appendix C, the Lex specification for identifying Ada language features is processed by the Lex program to generate a C program in a file called lex.yy.c. This file is then compiled to make an executable routine by itself or to interface to other programs. For some unknown reason, Lex includes reference to a C subroutine that is not included in lex.yy.c. When the object code generated from lex.yy.c is linked with the Ada programs, the Ada loader generates an error message. To eliminate this error message, it was necessary to manually remove the invalid reference from lex.yy.c before compiling it. The addition of a link name in the Verdix Ada library was required to tell the Ada linker where to find the Yylex object code. For the initial ALIANT prototype, the following process was developed to establish the Yylex to Ada interface (see Figure 4.7 and Figure 4.8):

1. The Lex specification in lex_spec is used to produce the yylex source code in lex.yy.c by executing the following command: "lex lex_spec".
2. The C source code in lex yy.c is modified to remove the undefined reference to yywrap (see Figure 4.9).
3. The modified C source code in lex.yy.c is compiled as follows to produce linkable object code in lex.yy.o: "cc -c lex.yy.c".
4. The file lex.yy.o is renamed yylex and the following entry is manually inserted into the Verdix Ada ada.lib file: "WITH1:LINK:yylex:". This entry tells the Ada linker where to find the yylex program mentioned in the Lex_Pkg using pragma interface.
5. The compiled Ada packages and driver are linked together with the yylex routine using the Verdix Ada command "a.ld aliant_driver -o aliant_driver.exe". The output name was arbitrarily chosen to easily identify the executable driver file.

Having resolved the technical issues just mentioned, the next iteration of the ALIANT prototype development led to the creation of the Features_Pkg. In the first prototype, the Display_Matrix procedure was just displaying the feature numbers and associated count totals. To translate the feature numbers back into the more informative Ada

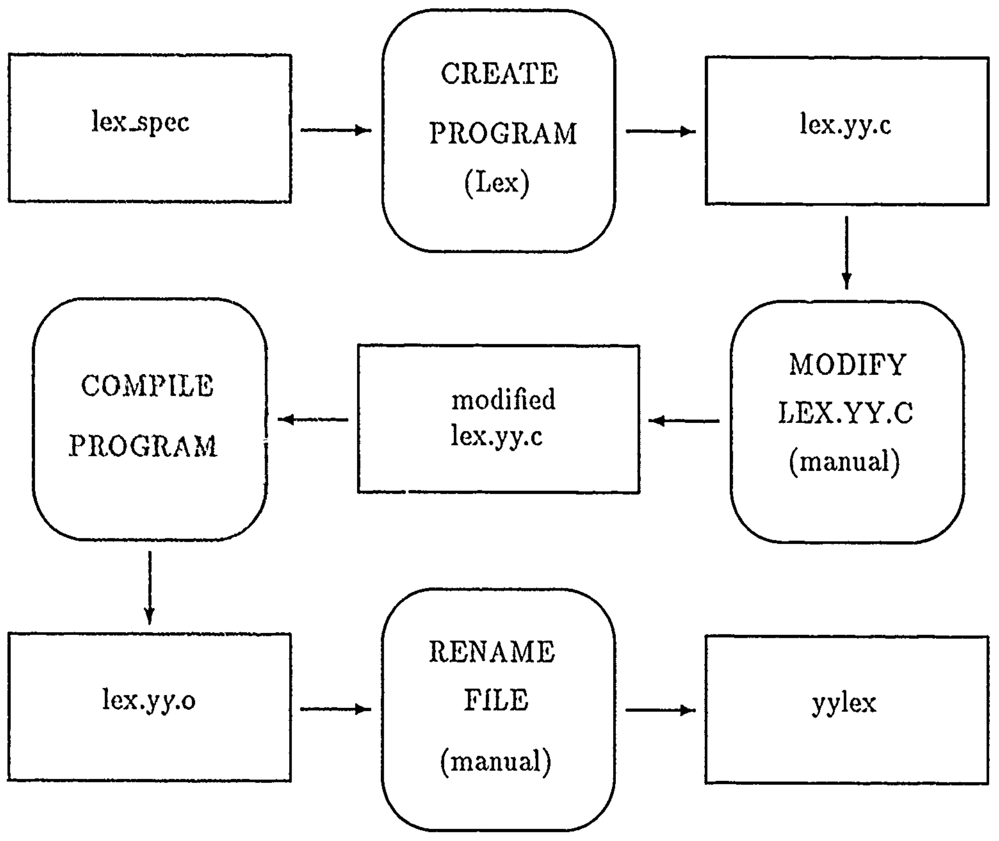

Figure 4.7. Creating Yylex Object Code

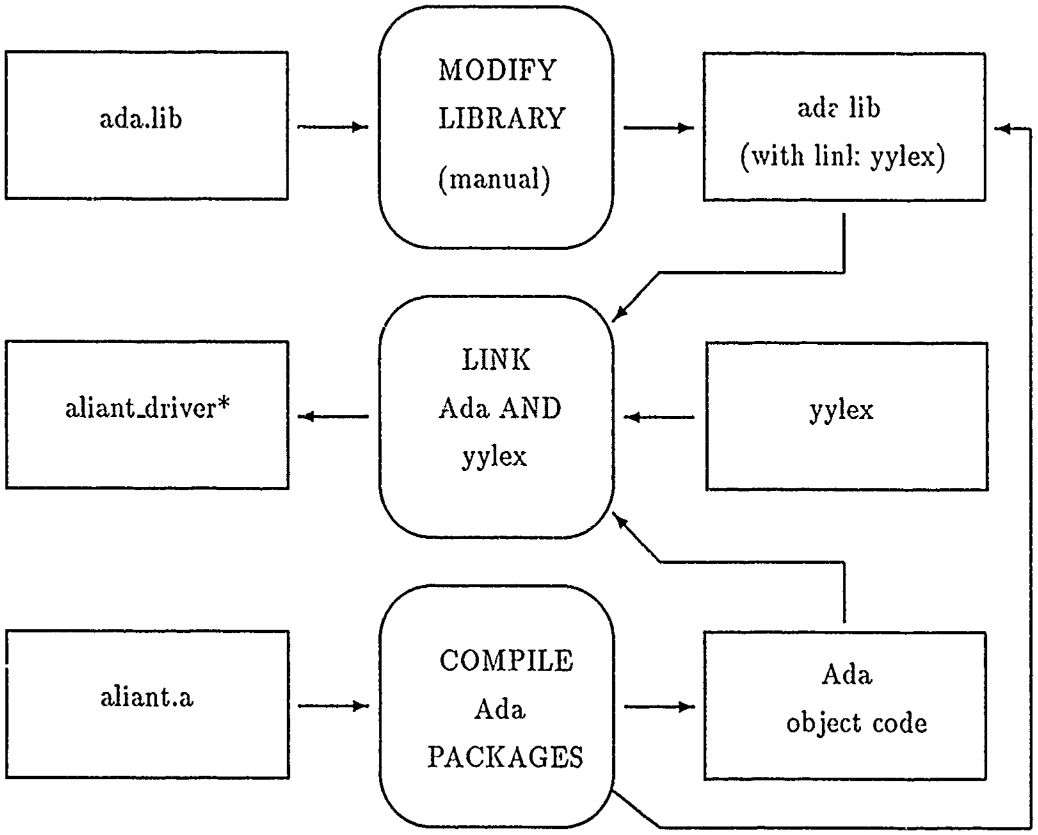

Figure 4.8. Creating Executable ALIANT_Driver

```
.
.
.
int yyleng; extern char yytext[];
int yymorfg;
extern char *yysptr, yysbuf[];
int yytchar;                  /*******************************************/
FILE *yyin, *yyout ={stdout}; /** REMOVED '={stdin}' FROM yyin FOR ALIANT **/
extern int yyneno;            /*******************************************/
struct yysvf {
struct yywork *yystoff;
struct yysvf *yyother;
int *yystops;};
struct yysvf *yyestate;
extern struct yysvf yysvec[], *yybgin;
# define YYNEWLINE 10
/*********************************************************************/
/****************  ADDED opengen AND closegen FOR ALIANT  ***************/
/*********************************************************************/
opengen(){
yyin = fopen("gen_out", "r");
}

closegen(){
fclose(yyin);
}
/********************************************************************/

yylex(){
int nstr; extern int yyprevious;
while((nstr = yylook()) >= 0)
yyfussy: switch(nstr){
case 0:                       /****************************************/
 return(0); break;            /*** REMOVED 'if(yywrap())' FOR ALIANT ****/
case 1:                       /****************************************/
                    { return(  1); }
break;
case 2:
              { return(  2); }
break;
.
.
.
```

Figure 4.9. Modifications to lex.yy.c Program

nonterminal/pseudo-nonterminal symbols, a look-up table containing the desired information is required. Rather than hard-code the table in the Ada source code, procedures were added to read in the information directly from the Lex specification (lex_spec) that was used to create the Lex_Pkg.Yylex function. This method guarantees the look-up table matches the feature tokens being returned by the Yylex function. Whenever a new lex_spec and Yylex are generated, the Ada programs do not have to be recompiled; only re-linked with the Yylex code. The Features_Pkg is described in Figure 4.10.

Load_Features_Table This procedure extracts the nonterminal and pseudo-nonterminal symbols from the lex_spec file and stores them in an array of character strings.

Get_Feature This function accepts a feature number as a parameter and returns the corresponding feature character string from the features table.

Figure 4.10. Features_Pkg Procedure Descriptions

While developing the Features_Pkg, it became apparent that the creation of the lex_spec file (Appendix C.2) could be automated using another Lex specification. All that was needed was a Lex specification that would recognize the format of nonterminal and pseudo-nonterminal strings within the annotated grammar, and a driver program to generate the lex_spec file. Appendix C.3 shows the resulting specification called mk_lspec. Note that the C driver program is included in the same file to eliminate additional interface requirements. The process to create the lex_spec file is similar to the process used to create the Yylex program from lex_spec (see Figure 4.11):

1. The Lex specification and main driver in mk_lspec is used to produce the source code in lex.yy.c by executing the following command: "lex mk_lspec".
2. The C source code in lex.yy.c is compiled and linked as follows to produce an executable stand-alone program in a.out*: "cc lex.yy.c -ll".

3. The lex_spec file is created by executing the following command: "a.out* < adagen.gen > lex_spec". In this example, the annotated grammar in adagen.gen is redirected as input to a.out* and the output from a.out* is redirected to lex_spec.

To further enhance the flexibility of the ALIANT prototype, and defer the declaration of the combination matrix from compile time until run-time, methods were developed to input the maximum number of features and maximum number of combinations during program execution. The first way this was attempted was by adding interactive prompts within the body of the Parameter_Pkg. Appropriate pragmas were used to make sure the Parameter_Pkg was elaborated before any other compilation units that depended on it. Therefore, the values input in the Parameter_Pkg body would be available in data structure declarations throughout the ALIANT prototype. Unfortunately, this first attempt revealed another technical issue that had to be resolved: how to explicitly open and close the Gen output file.

The initial prototype's use of piping to transfer the Gen output data into the ALIANT_Driver prevented the user from inputting information from the keyboard. The alternative was to redirect the Gen output to a file that would be explicitly opened from the ALIANT_Driver. This would "free-up" the standard input for interactive keyboard entries. The eventual solution to this problem required further modifications to the lex.yy.c file that was used to create the Yylex function. The following additional lex.yy.c modifications, plus the addition of Opengen and Closegen to the Lex_Pkg, solved the standard input problem.

1. The previously modified C source code in lex.yy.c (generated from lex_spec) is further modified as shown in Figure 4.9. The initialization of file yyin to standard input is

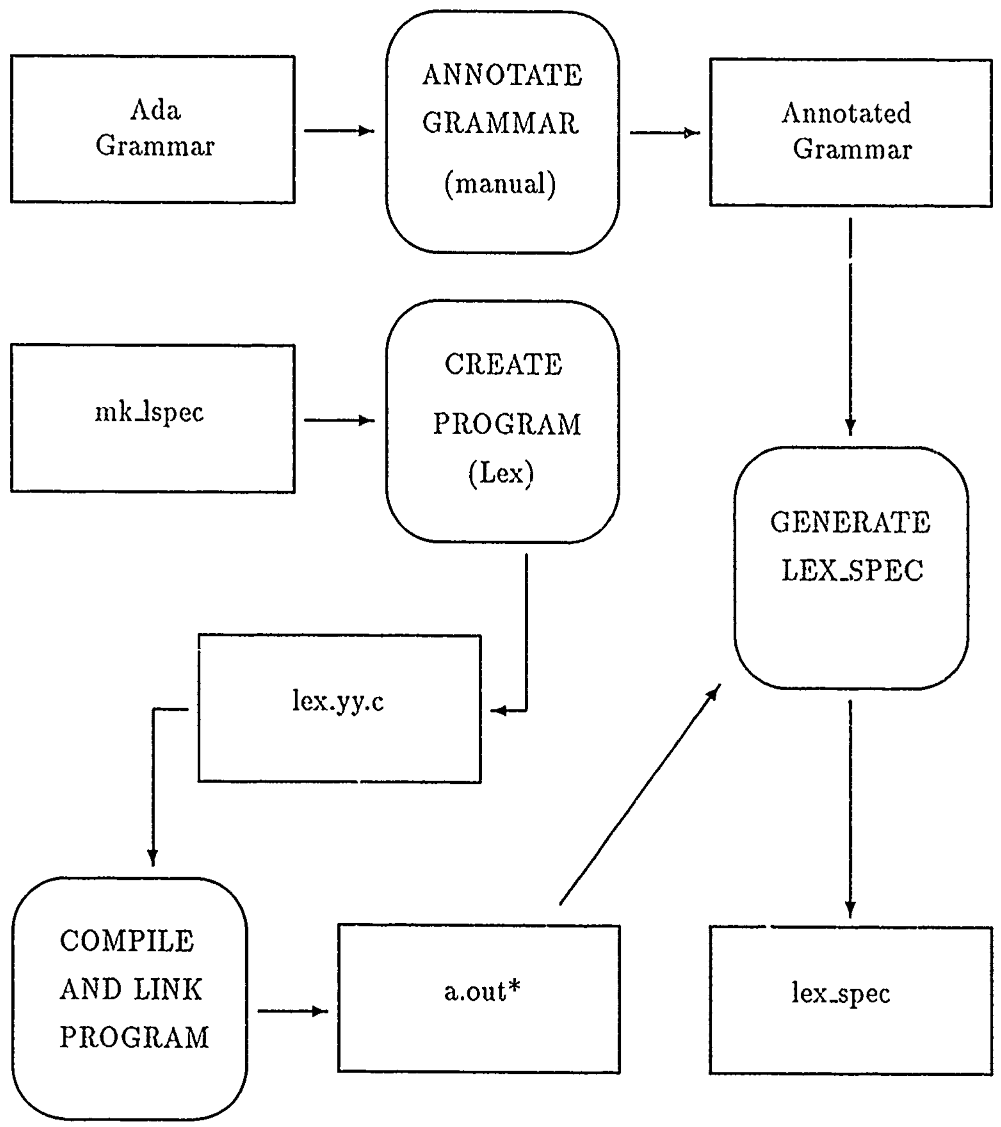

Figure 4.11. Creating lex_spec Lex Specification

changed so that yyin is uninitialized. Two subroutines, opengen and closegen, are added just before the yylex procedure.

2. The lex.yy.c is compiled as before with "cc -c lex.yy.c" to produce lex.yy.o.
3. The file lex.yy.o is renamed yylex as before and relinked with the Ada programs.

The interactive capability allowed the user to input the number of features and number of combinations at run-time, but a better method is to extract this information automatically from other sources. The lessons learned in developing the Features_Pkg were applied to the problem of getting the number of features without user input. By using a similar technique as the Load_Features_Table procedure, the Parameter_Pkg body was modified to read in the lex_spec file to count the number of features. This technique would further guarantee that the maximum number of features was synchronized with the Yylex function and the Features_Table. To get the number of combinations automatically required an interface file between the shell script and the Ada programs.

As described in the previous section on the Gen to ALIANT interface, the UNIX shell script input was eventually developed to allow the user to input a parameter for the number of Gen combinations to be generated. This enhancement led to an automatic technique to input the maximum number of combinations. The number of combinations that is used to generate the Gen test cases is passed to the ALIANT programs through the g_temp file. Note that the maximum combinations parameter is used to declare the size of the combination matrix. Therefore it should be as small as possible to contain the expected number of "unique" combinations. For example, of 5000 combinations generated using the Adagen1 grammar in Appendix E.1, 2512 combinations were null, while 2296 combinations were duplicates. This leaves only 192 combinations that had to be stored in

. the combination matrix. In this example, the required space is only 4 percent of the total number of combinations generated.

For very large test runs, the difference between the number of generated combinations and the number that are actually put in the matrix can be considerable. Therefore, the number in the g_temp file is much larger than the expected number of combinations to be stored. By observing several test runs over a wide range of combination numbers, a mathematical function was developed that calculates an approximation for the maximum number of combinations. The calculation is based on the number input from g_temp. Table 4.1 shows a comparison of the estimated versus actual number of stored combinations; where x is the number input from g_temp, and y is the estimate used to determine one dimension of the combination matrix. Two different values of the coefficient c are shown. For the simple grammar shown in Appendix E.1, a coefficient of 17 is adequate. But for the more complex Adagen2 grammar of Appendix E.2, in which there are no null combinations generated, a coefficient of 1200 is required. Although the larger coefficient results in "wasted" space for smaller numbers of combinations, it still yields a significant space savings for the larger numbers. Instead of developing a much more complicated function calculation to obtain closer approximations for all values, the simple $y = \sqrt{cx}$ function was adopted for demonstration purposes in this prototype. For this particular grammar, the estimate is sufficient up to about 10,000 combinations. A grammar of different complexity or larger numbers of combinations may require an increase in the x coefficient to make sure the combination matrix is large enough.

With the introduction of the interactive capability, several new features were iteratively added as the ALIANT prototype evolved. The most significant of these added

Table 4.1. Estimated vs. Actual Number of Combinations

$y = \sqrt{cx}$				
g_temp	$c = 17$	Adagen1	$c = 1200$	Adagen2
100	41	38	346	69
200	58	54	489	122
300	71	68	600	171
400	82	77	692	217
500	92	84	774	264
600	100	91	848	296
700	109	96	916	342
800	116	100	979	389
900	123	107	1039	433
1000	130	112	1095	480
2000	184	139	1549	827
3000	225	164	1897	1192
4000	260	178	2190	1521
5000	291	192	2449	1847
10000	412	218	3464	3452

features was the ability to specify *thresholds* for selective displaying/storing of the generated feature combinations. In response to on-screen prompts, the user of the ALIANT prototype may choose to display/store feature combinations based on the number of duplicates each combination had and the number of features in each combination. The user is prompted to enter the two threshold values that will be used to select the desired combinations. Any combination that has an *equal or greater* number of duplications as the duplication threshold value, or an *cqual or less* number of features as the feature threshold, is chosen for display or loading into the database. While in an ALIANT session, the user may repeatedly try different threshold values to see how many combinations are chosen. The number of combinations that satisfy the specified threshold values is displayed first. Then the user is given the opportunity to view the combinations with or without on-screen paging, or not at all. After the user is finished selecting/displaying combinations, he/she may select combinations to be loaded into the AFIS database file. The thresholds are used as before to select which combinations are output to the database file. If the batch input option is used to execute an ALIANT session, the desired sequence of user inputs is stored in an ASCII input file and redirected into ALIANT from within the UNIX shell script. Appendix F gives detailed operating instructions and screen display examples of the ALIANT prototype.

The ALIANT prototype expects several executable/data files to be available in the same directory. In some cases, the *versions* of these files must also agree. For example, the Yylex subroutine is generated for a particular set of features in an annotated grammar. If the grammar is modified to include additional features, they will not be recognized as valid, until a new Yylex program is generated. If certain files are missing, the standard

Ada run-time abort messages would not clearly indicate where the problem occurred. Therefore, several exception conditions were identified and descriptive error messages were developed to guide the user to the cause of some of the common problems that could arise. In addition to the errors detected within the Ada programs, the UNIX shell script does preliminary error checking of the input parameters to avoid unnecessary execution of the ALIANT prototype. Figure 4.12 summarizes the types of errors that are detected by the entire prototype.

This chapter presented a description of the iterative method used to develop the ALIANT prototype. Initial versions helped confirm necessary interfaces would work. Later versions added the full functionality described in the original design of the previous chapter. The next chapter will analyze the operation of the ALIANT prototype.

Wrong number of input arguments If too many, or too few, input arguments are provided to the shell script, an error message is generated showing the proper input format.

Non-existent filenames for input arguments If a non-existent Gen input file or batch input file is provided, an error message is generated by the shell script and execution is terminated.

Invalid number of combinations If the argument for the number of combinations to generate is not greater than zero, the shell script generates an error message and terminates execution.

Gen program abort If the Gen program does not terminate normally, the ALIANT_Driver is not allowed to execute. This prevents analysis of partial or non-existent Gen output.

Errors concerning the lex_spec file If the lex_spec file is non-existent or the contents do not match the expected format, error messages are generated from within the ALIANT code. The lex_spec file is opened during the initialization of the Parameter_Pkg body and during the Load_Features_Table execution. The generated error messages indicate at which location the error occurred.

Errors concerning the g_temp file Although unlikely, the g_temp file may contain errors when it is opened by the ALIANT code. If so, error messages are generated for a non-existent file or one in which the number of combinations is not in the expected format. Error messages are also generated if the number of combinations exceeds the predefined constraint range.

Too many generated combinations Since a function is used to calculate the *expected* matrix storage requirements, it is possible that the space will be exceeded. If this happens, an error message is generated before the partial matrix of combinations is made available for user display.

Errors concerning the gen_out file If the format of the gen_out file from Gen has been contaminated, an error message is generated. This error message is only generated if a feature is identified before the start combination symbol is identified.

User input errors If the user inputs invalid threshold values, an error message is briefly displayed on the screen and then another input prompt is provided. If the user inputs come from a batch file, it is possible the wrong number of inputs are provided and an end-of-file condition will be reached on standard input. If this occurs, a specific error message is generated indicating the possible cause.

Undefined feature token If a Gen input grammar is used that contains new features that are not in the current Yylex routine, an informational error message will be generated while the combinations are analyzed.

Figure 4.12. ALIANT Prototype Error Detection

V. Test and Analysis

The purpose of this chapter is to analyze the results of ALIANT prototype testing. First, the overall test objectives are presented. Then, the grammar test procedures are described. Next, the test results, using two versions of the annotated Ada grammar, are presented. The test data show how the degree of grammar complexity affects the number and type of feature combinations the prototype produces. Finally, the results of error-checking analysis are provided. This analysis verifies the prototype responds to error conditions as designed; providing the user with an indication of how the problem can be corrected.

5.1 Test Objectives

The primary objectives of ALIANT prototype testing are to

- Verify the prototype is correctly generating and identifying Ada feature combinations according to the input annotated grammar.
- Gather performance statistics for ALIANT execution using two versions of a grammar and a range of values for numbers of combinations, duplication thresholds, and feature thresholds.
- Verify the prototype error detection and message display features work as designed.

The first objective is not explicitly documented elsewhere in this chapter. Initial tests were accomplished in which the Gen output was visually inspected to verify the proper output format. Other independent tests were used to confirm the randomness constructs did produce the desired selection of alternatives. For example, given a production in which one alternative is to be chosen 30 percent of the time, the tests confirmed the specified alternative did appear 30 percent of the time. The proportions were more accurate for larger numbers of generated combinations. Other tests were conducted to manually compare the ALIANT processing statistics and selected combinations with the actual Gen output. These tests confirmed the Ada code was correctly counting features, identifying null and duplicate combinations, etc.

The second test objective was designed not only to gather data for later analysis, but also to "stress test" the ALIANT prototype. By using grammar versions of different complexity, and a wide range of combination numbers and threshold values, any prototype deficiencies should be uncovered. The values used for testing were designed to cover the expected boundary values, as well as selected values in between. The maximum number of combinations tested, 10000, was chosen somewhat arbitrarily due to the ALIANT execution time and memory requirements. These larger test runs took over an hour to complete. The chosen combination numbers were sufficient to establish clear trends for analysis.

The third test objective was satisfied by setting up a specific error condition for each category of ALIANT error detection capability. The final section of this chapter presents the results of these specific tests.

5.2 Grammar Test Setup

Two versions of an annotated Ada grammar were chosen for testing the ALIANT prototype (see Appendix E). As described in the previous chapter, the first grammar (Adagen1) is annotated very simply. Any productions that had terminal symbols on the right hand sides were reduced to a character string in quotation marks. No weighting values were assigned to any of the Gen randomness constructs, which made any alternative equally likely in a given production. The second grammar (Adagen2) is more complex. This grammar was annotated to allow the generation of more detailed feature combinations. The 297 *primary features* used as a preliminary "filter" to the Program Analysis Tool (PAT) were used as a guide in annotating the Adagen2 grammar (the *primary features* were discussed in Chapter I). More realistic alternative probabilities were added to the Gen randomness constructs to guide the generation of test cases. The added probabilities were "more realistic" in the sense that more commonly used Ada features were given a higher probability of occurring. The choices for the alternative probabilities are based on the subjective judgement of the individual annotating the grammar. The use of objective statistical data would result in the "most realistic" grammar annotation.

The testing process with these grammars was accomplished by using the ALIANT batch input option. A batch file was set up to accumulate test data for 11 different pairs of *duplicate* and *feature* threshold values. Recall that the *duplicate* threshold indicates the minimum number of duplicates a combination must have to qualify for selection, while the *feature* threshold indicates the maximum number of features a combination can contain to qualify for selection. The 11 test selections are shown in Table 5.1. The first five selections, A through E, were designed to show how the number of combinations changed by duplication threshold, while the feature threshold was constant. The feature threshold was maintained at 100, to avoid eliminating combinations based on that factor, as the duplication threshold was varied from 4 to 0. The next five test selections, F through J, were designed to show the effect of changing the feature threshold value. The duplication threshold of 0 made sure no combinations were eliminated due to that factor while the feature threshold was varied from 50 to 10. The final test option, K, was chosen to illustrate a selection of recommended feature combinations to test. This selection chooses those combinations that have a duplicate count greater than or equal to 2, and a feature count less than or equal to 20. In other words, a combination of 20 features that occurred 3 times would be selected; but a combination of 20 features that occurred only 2 times would not be selected.

A UNIX shell script was created to execute the batch input for 15 different "numbers of combinations", ranging from 100 to 10000 combinations. In other words, the ALIANT batch input file containing the appropriate user input entries for the 11 different pairs of threshold values was executed first for 100 generated combinations, then 200 combinations, and so on up to 10000 combinations (after 1000 combinations, the successive tests increased 1000 combinations at a time). The shell script automatically moved the output from "alnt_out" to a unique filename so the information from one test run would not be overwritten by subsequent test runs. The resulting data for all test selections and input combinations are presented in the following sections.

5.3 Adagen1 Grammar Results

The tabulated test data for the Adagen1 grammar is provided in Table 5.2. The first column is the number of *requested* combinations input to the ALIANT prototype. The next eleven columns, A through K, contain the number of combinations that satisfied each pair of test selection criteria described in the previous section. The final three columns contain the ALIANT summary totals for null, duplicate, and resulting combinations.

Since the numbers in columns E through J are constant, the data show that Adagen1 does not produce any combinations containing more than 10 features. If there was at least

Table 5.1. ALIANT Test Selection Options

TEST SELECTIONS		
Selection	Duplicate Threshold	Feature Threshold
A	4	100
B	3	100
C	2	100
D	1	100
E	0	100
F	0	50
G	0	40
H	0	30
I	0	20
J	0	10
K	2	20

Table 5.2. Adagen1 Grammar Test Results

Adagen1 Test Data							
	Test Selections						
# Comb	A	B	C	D	E	F	G
100	1	2	4	11	38	38	38
200	5	9	14	19	54	54	54
300	7	13	20	36	68	68	68
400	12	17	24	43	77	77	77
500	17	21	30	49	84	84	84
600	18	24	37	56	91	91	91
700	21	26	43	59	96	96	96
800	23	33	48	62	100	100	100
900	25	35	50	66	107	107	107
1000	29	38	53	70	112	112	112
2000	53	63	77	98	139	139	139
3000	70	78	91	116	164	164	164
4000	80	93	107	128	178	178	178
5000	88	102	115	141	192	192	192
10000	122	131	158	182	218	218	218

Table 5.2. Adagen1 Grammar Test Results (continued)

Adagen1 Test Data (continued)							
	Test Selections				Summary Totals		
# Comb	H	I	J	K	Null	Dup	Res
100	38	38	38	4	43	19	38
200	54	54	54	14	95	51	54
300	68	68	68	20	140	92	68
400	77	77	77	24	194	129	77
500	84	84	84	30	243	173	84
600	91	91	91	37	293	216	91
700	96	96	96	43	349	255	96
800	100	100	100	48	398	302	100
900	107	107	107	50	444	349	107
1000	112	112	112	53	494	394	112
2000	139	139	139	77	1008	853	139
3000	164	164	164	91	1509	1327	164
4000	178	178	178	107	2011	1811	178
5000	192	192	192	115	2512	2296	192
10000	218	218	218	158	4972	4810	218

one combination with 11 features, the numbers in columns E through I would be one greater than the number in column J. In fact, additional tests revealed the maximum number of features per combination is 9. A visual examination of the Adagen1 grammar in Appendix E.1 confirms that only 9 features are produced by the following 21 "reachable" productions:

```
subprogram_declaration = ( subprogram_specification )
subprogram_specification = ( "subprogram_specification " )
subprogram_body = ( "subprogram_body " )
package_declaration = ( package_specification )
package_specification = ( "package_specification " )
package_body = ( "package_body " )
use_clause = ( "use_clause " )
compilation = ( "START_COMPILATION: "
  ( "" | % compilation_unit more_units ) ":END_COMPILATION \n" )
more_units = ( "" | % compilation_unit more_units )
compilation_unit = (
  context_clause library_unit
  | % context_clause secondary_unit )
library_unit = (
  subprogram_declaration | % package_declaration
  | % generic_declaration | % generic_instantiation
  | % subprogram_body )
secondary_unit = ( library_unit_body | % subunit )
library_unit_body = ( subprogram_body | % package_body )
context_clause = (
  "" | % ( with_clause ( "" | % use_clause more_use ) context_clause ) )
more_use = ( "" | % use_clause more_use )
with_clause = ( "with_clause " )
subunit = ( "subunit " )
generic_declaration = ( generic_specification )
generic_specification = (
  generic_formal_part subprogram_specification
  | % generic_formal_part package_specification )
generic_formal_part = ( "generic_formal_part " )
generic_instantiation = ( "generic_instantiation " )
```

The "reachable" productions form a tree with the *compilation* production as the root and the terminal productions as the leaves. Since a *compilation* is composed of zero or more *compilation_units*, all 9 features can appear in a single combination.

The relative simplicity of Adagen1 combinations is indicated by the fact that there are only 218 different combinations produced from 10000 attempts. Figure 5.1 gives a graphic illustration of what is happening as more and more combinations are generated. The graph plots the ALIANT summary totals for null, duplicate, and resulting combinations as well as selected combinations for a specified pair of threshold values. Selected combinations are usually a subset of all the resulting combinations if the threshold values are chosen properly. Note that a duplication threshold equal to zero, and a feature threshold equal to the maximum number of features will guarantee all resulting combinations are selected. The null and duplicate combinations are growing at a steep linear rate, while the resulting combinations are increasing much slower. Theoretically, the resulting combinations should eventually reach a maximum or at least approach a maximum value. The decreasing slope of the resulting combinations is an indication that such a maximum does exist. The plotted slope of the selected combinations is decreasing as well. For the indicated selection thresholds (Dup-2 Feature-20), the graph shows a much slower growth rate for a subset of all generated combinations. The absolute maximum number of combinations could be calculated, but the ALIANT prototype would spend a lot of time generating mostly duplicates while attempting to exhaust all possibilities. For the rather simple Adagen1 grammar, it is likely that exhaustive generation of all possible feature combinations could be achieved. Unfortunately, to be of any value to the ACVC testing effort, much more complicated grammars must be annotated. The next section shows the corresponding test results for the Adagen2 grammar.

Before presenting the Adagen2 testing results, an additional Adagen1 demonstration is provided. The output combinations produced by Adagen1 can be reduced further by modifying the *compilation* and *more_units* productions from this format:

```
compilation = ( "START_COMPILATION: "
  ( "" | % compilation_unit more_units ) ":END_COMPILATION \n" )
more_units = ( "" | % compilation_unit more_units )
```

to this format:

```
compilation = ( "START_COMPILATION: "
  ( "" | % 0 compilation_unit more_units ) ":END_COMPILATION \n" )
more_units = ( "" | % 100 compilation_unit more_units )
```

The addition of the randomness percentages 0 and 100 as indicated cause the grammar to generate one *compilation_unit* per combination. Figure 5.2 shows the resulting data for this modified Adagen1 grammar called "T-Adagen1". Note that the duplicate combinations continue to grow at a higher rate than before, while the resulting combinations reaches a peak value very early. In this case, the grammar can only generate a maximum of 24 different *compilation_units*. This actual maximum for the 9 features of the 21 "reachable" productions in T-Adagen1 is much less than an easily calculated *least upper bound* of 511. The least upper bound is determined by calculating the number of combinations of 9 features taken 1 at a time, 2 at a time, and so on up to 9 at a time. The sum of these calculations, 511, is the least upper bound for this grammar. The reason the actual

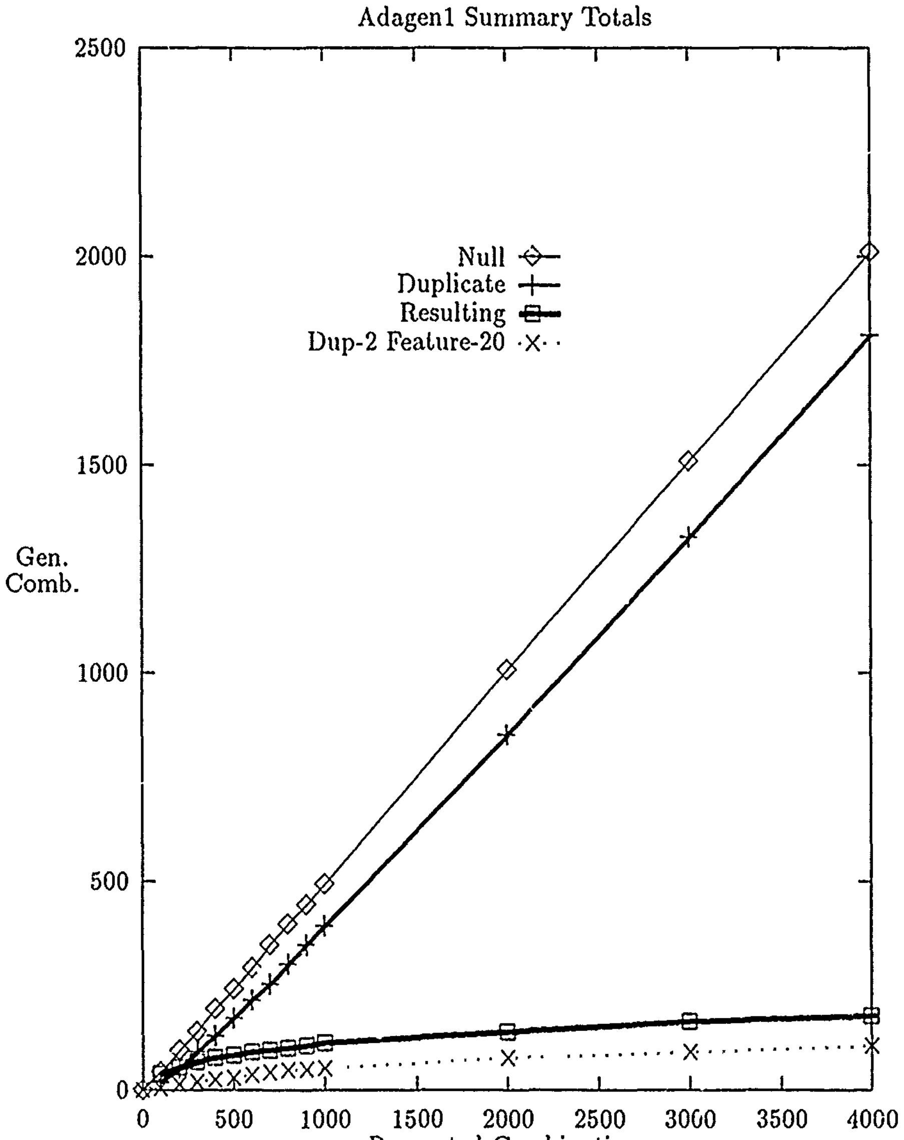

Figure 5.1. Adagen1 Summary Totals Graph

maximum number of combinations is much less than the calculated upper bound is that many of the calculated combinations are not possible in actual practice. For example, a *with_clause* cannot appear alone in a combination.

5.4 Adagen2 Grammar Results

The tabulated test results for Adagen2 are provided in Table 5.3. The data is presented in the same manner as for Adagen1. The Adagen2 grammar did not generate any null combinations since the randomness constructs were modified in the *compilation* production. Rather than generate null combinations that ALIANT would only discard, the Adagen2 grammar eliminated the *zero* option by using zero percent on the corresponding alternative symbol. The elimination of null combinations causes many more *resulting* combinations to be generated.

The added complexity of the Adagen2 grammar is apparent from the data in columns F through J. The data in these columns indicate there are at least some combinations with more than 50 features. Additional tests identified at least one combination with as many as 67 features. Figure 5.3 shows the summary totals for the Adagen2 duplicate, resulting, and selected combinations. The growth rate for all but the selected combinations (Dup-2 Feature-20) is much greater than what was noted for Adagen1. This indicates a theoretic maximum is well beyond the reach of practicality, which was a premise for this research. In other words, it is not practical to test *all* feature combinations. The growth rate for the selected combinations is very similar to corresponding graph for Adagen1. The growth rate for this selected subset of all Adagen2 combinations is much lower than the overall rate.

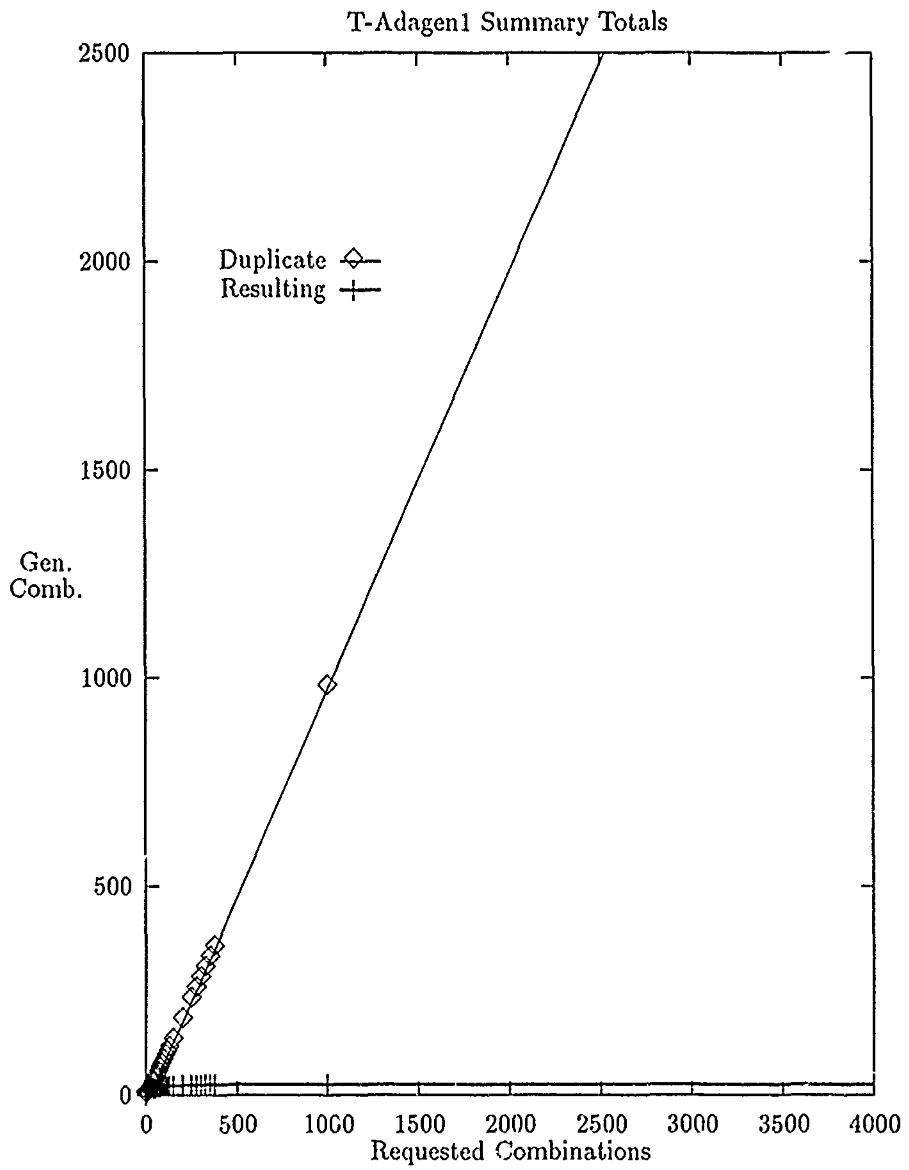

Figure 5.2. T-Adagen1 Summary Totals Graph

Table 5.3. Adagen2 Grammar Test Results

Adagen2 Test Data							
	Test Selections						
# Comb	A	B	C	D	E	F	G
100	1	4	10	16	69	69	67
200	9	11	16	27	122	122	119
300	12	18	25	36	171	170	164
400	18	21	29	39	217	216	208
500	20	27	33	45	264	262	254
600	26	31	39	47	296	294	285
700	29	34	43	54	342	339	328
800	30	35	46	57	389	386	373
900	32	37	50	66	433	430	416
1000	35	40	51	69	480	477	460
2000	56	65	83	120	827	817	790
3000	69	79	105	156	1192	1181	1133
4000	79	95	132	196	1521	1508	1438
5000	92	110	160	221	1847	1832	1748
10000	156	178	229	322	3452	3418	3247

Table 5.3. Adagen2 Grammar Test Results (continued)

Adagen2 Test Data (continued)							
	Test Selections				Summary Totals		
# Comb	H	I	J	K	Null	Dup	Res
100	62	47	43	10	0	31	69
200	106	81	68	16	0	78	122
300	147	111	92	25	0	129	171
400	187	146	121	29	0	183	217
500	227	174	144	33	0	236	264
600	251	190	155	39	0	304	296
700	281	214	171	43	0	358	342
800	321	243	191	46	0	411	389
900	354	269	211	50	0	467	433
1000	395	300	234	51	0	520	480
2000	675	498	377	83	0	1173	827
3000	965	705	522	105	0	1808	1192
4000	1207	853	606	132	0	2479	1521
5000	1466	1013	714	160	0	3153	1847
10000	2680	1773	1183	229	0	6548	3452

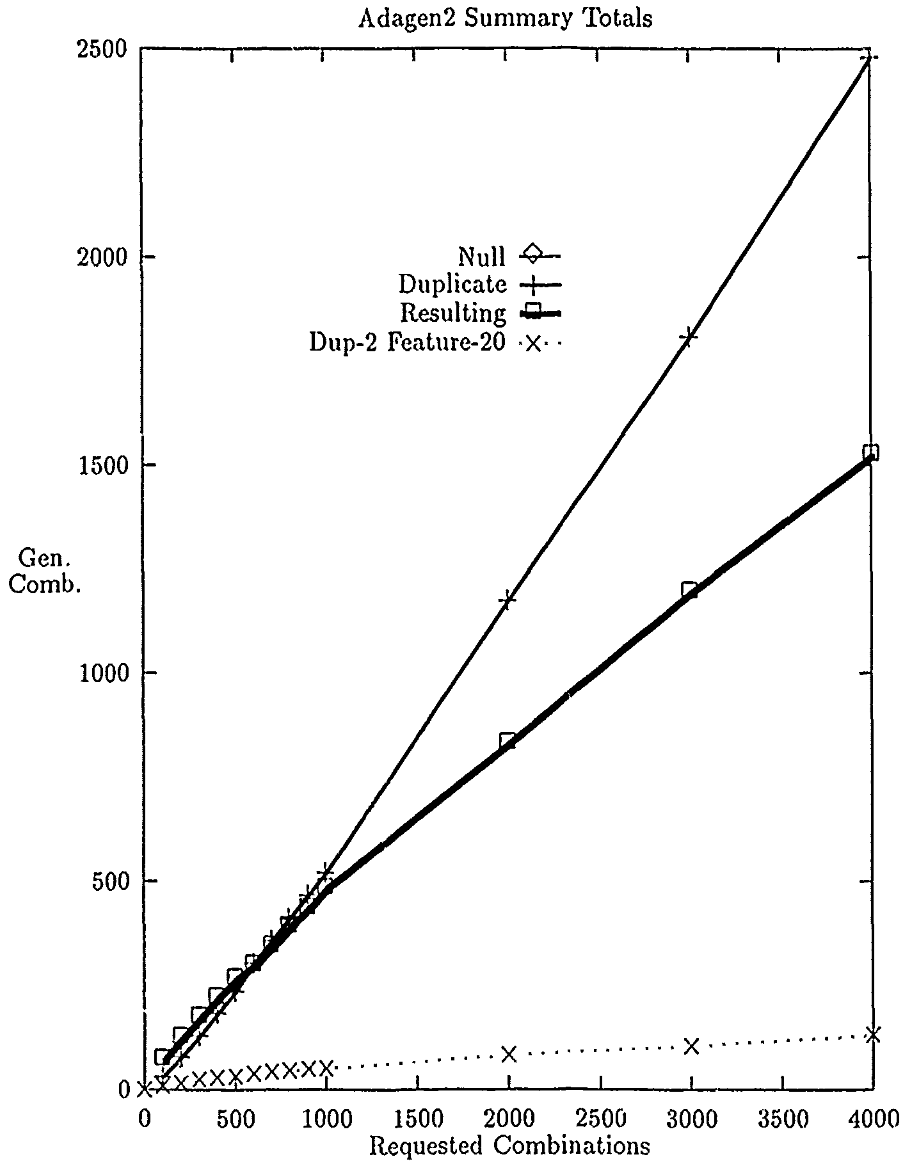

Figure 5.3. Adagen2 Summary Totals Graph

As was done with Adagen1, a modified version of Adagen2 called "T-Adagen2" was tested. This version of Adagen2 produces only one *compilation_unit* per combination. Figure 5.4 shows the results. The *resulting* combinations are still growing at a much higher rate than Adagen1. Even with the reduced complexity of each combination, the overall grammar complexity still makes the maximum value impractical to reach. Note that the original Adagen2 grammar already had the *more_units* alternative percentage set at 90 percent which caused the null alternative to be chosen 90 percent of the time. Therefore, setting this value to 100 percent did not result in a dramatic change between Adagen2 and T-Adagen2, as was noted between Adagen1 and T-Adagen1.

5.5 *Error Condition Analysis*

The normal operation of the ALIANT prototype was thoroughly tested while collecting test data discussed so far in this chapter. Although some of the error-checking features were verified during the course of this data collection, several special tests were required to be sure all such features were working properly. The purpose of this section is to present the results of a series of tests that confirmed the prototype error-checking worked as expected. In each case, the method used to produce an error is described, followed by the results of the induced error.

- Invalid parameters for *runa* shell script.
  TEST 1: The *runa* shell script was invoked with no parameters.
  RESULT: The following error message was displayed before the prototype was terminated:

```
*
* Missing Gen filename and/or combinations argument(s), try again.
*
```

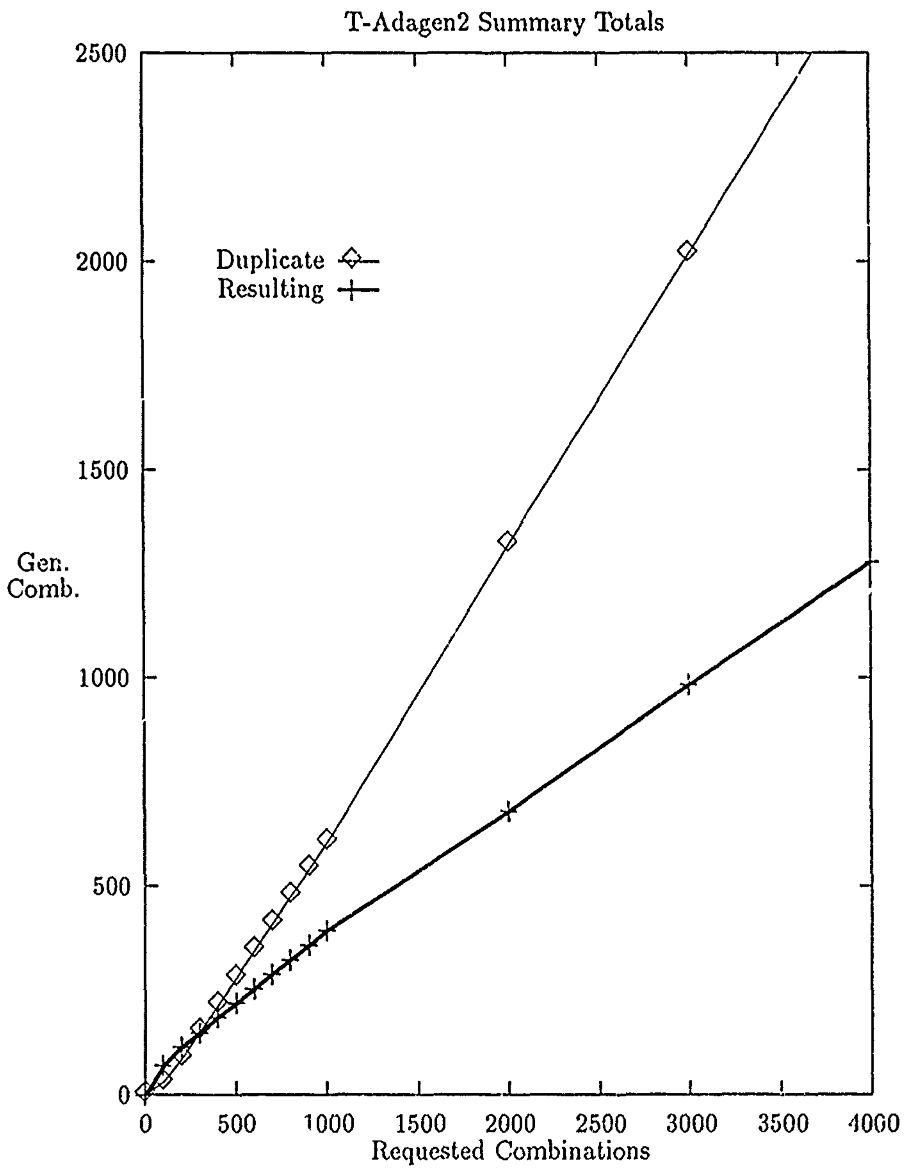

Figure 5.4. T-Adagen2 Summary Totals Graph

```
*---- Format: runa* fn1 num [fn2]
*---- Where : fn1.gen is the Gen input file,
*------------ num is the desired number of combinations, and
*------------ fn2 is an optional ALIANT batch input file.
*
```

TEST 2: A non-existent filename was provided for the Gen input file.

RESULT: The following error message was displayed before the prototype was terminated:

```
*
* Gen filename provided does not exist and/or number
* of combinations provided not greater than 0, try again.
*
```

- Gen program abort.

  TEST 3: The *runa* shell script was invoked with valid parameters; but the Gen executable file was not in the current directory. This test simulates any condition in which the Gen software aborts before or during execution.

  RESULT: The operating system produced an error message and the shell script terminated execution without running the ALIANT driver program.

```
--------------------------------
-- Gen execution in progress --
--------------------------------
gen.exe*: No match.
```

- Errors concerning the lex_spec file.

  TEST 4: The ALIANT prototype was executed with a non-existent lex_spec file.

  RESULT: The following error message was produced during execution of the Parameter_Pkg initialization. Note that an Ada runtime abort message was produced since the ALIANT_Driver had not been elaborated yet. Normally, the ALIANT_Driver will replace the system message with an ALIANT terminating message.

```
 <Parameter_Pkg body>
 *** NAME EXCEPTION ERROR RAISED WHILE  ***
 *** TRYING TO OPEN LEX SPECIFICATION   ***
 *** FILE.  CHECK FILENAME IN PARAMETER ***
 *** PACKAGE AND CURRENT DIRECTORY.     ***

** MAIN PROGRAM ABANDONED -- EXCEPTION "FATAL_EXCEPTION" RAISED
```

  TEST 5: The ALIANT prototype was executed with an empty lex_spec file. This test simulates a condition in which the lex_spec is present but contains format errors.

  RESULT: The following error message was produced during execution of the Parameter_Pkg initialization. As in the previous test, an Ada runtime abort message was also produced.

```
<Parameter_Pkg body>
*** PREMATURE END-OF-FILE REACHED WHILE ***
*** READING LEX SPECIFICATION FILE.     ***
*** CHECK FORMAT OF LEX SPECIFICATION.  ***

** MAIN PROGRAM ABANDONED -- EXCEPTION "FATAL_EXCEPTION" RAISED
```

- Errors concerning the g_temp file.

  TEST 6: The ALIANT_Driver is executed manually (without using the shell script) with a non-existent g_temp file. This test demonstrates a condition that is unlikely when using the shell script; but very likely if the ALIANT_Driver is used to process a previously generated Gen output file.

  RESULT: The following error messages were produced before the prototype was terminated:

```
<Parameter_Pkg body>
*** NAME EXCEPTION ERROR RAISED WHILE  ***
*** TRYING TO OPEN THE GEN COMBINATION ***
*** FILE.  CHECK FILENAME IN PARAMETER ***
*** PACKAGE AND CURRENT DIRECTORY.     ***

** MAIN PROGRAM ABANDONED -- EXCEPTION "FATAL_EXCEPTION" RAISED
```

  TEST 7: The ALIANT_Driver is executed manually with a null g_temp file.

  RESULT: The following error messages were displayed before the prototype was terminated:

```
<Parameter_Pkg body>
*** PREMATURE END-OF-FILE REACHED WHILE ***
*** READING GEN COMBINATION FILE. CHECK ***
*** FORMAT OF GEN COMBINATION FILE.     ***

** MAIN PROGRAM ABANDONED -- EXCEPTION "FATAL_EXCEPTION" RAISED
```

  TEST 8: The ALIANT prototype was executed with a "number of combinations" parameter that was larger than the hard-coded Parameter_Type in the Parameter_Pkg.

  RESULT: The following messages were displayed before the prototype was terminated:

```
<Parameter_Pkg body.2>
*** NUMBER OF COMBINATIONS IS OUT OF      ***
*** RANGE. CHECK GEN COMBINATION FILE AND ***
*** PARAMETER TYPE IN PARAMETER PACKAGE.  ***

** MAIN PROGRAM ABANDONED -- EXCEPTION "FATAL_EXCEPTION" RAISED
```

- Too many generated combinations.

  TEST 9: The ALIANT prototype was executed with a "number of combinations" parameter that would cause the calculated Max_Combinations to be exceeded.

  RESULT: The following message was displayed before the remaining ALIANT screenfaces were presented.

```
<Matrix_Pkg.Start_Combination>
*** TOO MANY GEN COMBINATIONS. ***
*** PARTIAL RESULTS FOLLOW.    ***
```

- Errors concerning the gen_out file.

  TEST 10: The ALIANT prototype was executed manually with a gen_out file that contained an error. The invalid gen_out file contained a valid feature but was missing the "START_COMPILATION:" string.

  RESULT: The following error messages were displayed before the prototype was terminated:

```
<Matrix_Pkg.Count_Feature>
*** INCORRECT FORMAT IN GEN INPUT ***
*** FILE.  CHECK THE GEN GRAMMAR. ***

**************************************************
** Exiting ALIANT driver due to fatal exception **
**************************************************
```

- User input errors.

  TEST 11: During an interactive ALIANT execution, an invalid threshold value was entered (non-natural number).

  RESULT: The following error message was temporarily displayed on the screen before the user innut prompt was redisplayed:

```
** INVALID THRESHOLD VALUE -- MUST BE A NATURAL NUMBER **
```

  TEST 12: The ALIANT prototype is executed with a batch input file that does not contain enough entries for the input prompts that will be produced.

  RESULT: The end of the alnt_out file contained the following error messages:

```
<Matrix_Pkg.Display_Matrix.1>
*** END-OF-FILE REACHED ON STD INPUT. ***
*** PROBABLY INVALID ENTRIES IN THE   ***
*** ALIANT BATCH INPUT FILE (IF USED).***

**************************************************
** Exiting ALIANT driver due to fatal exception **
**************************************************
```

- Undefined feature token.

  TEST 13: The ALIANT prototype is executed with a lex_spec file that does not match the input grammar.

  RESULT: The following error message was displayed each time an unknown character string was identified.

```
** Undefined Token # 997, regenerate lex_spec file from input grammar. **
```

The results for all the error-checking tests are correct. In most cases, an error message gives the user a suggestion how the error might be corrected. When such diagnosis is not possible, the response is designed to prevent a user from receiving corrupted data. While all possible errors cannot be detected, the most likely errors are detected and the user notified.

The test results presented in this chapter have demonstrated the ALIANT prototype is operating as designed. The next chapter will discuss how well the ALIANT prototype satisfies the original requirements and offer recommendations for further research.

VI. Conclusions and Recommendations

The original requirement for the ALIANT prototype was to develop a system that would automatically identify recommended Ada compiler test combinations. The scope was limited to the identification of the recommended combinations of Ada features. There was no attempt to generate compilable test cases containing the recommended combinations. This chapter will discuss how well the ALIANT prototype meets the original requirement and will make recommendations for further research in this area.

6.1 Research Conclusions

The ALIANT prototype can provide a valuable service to ACVC support personnel. Using the Adagen2 grammar (Appendix E.2) as is, or with minor modifications, AMO personnel can use the prototype to identify potential feature combinations to be tested. Since the Adagen2 grammar is already annotated with virtually all of the 297 *primary features*, the ALIANT prototype output can be interfaced with the existing Program Analyzer Tool (PAT). Since the *duplication* and *feature* threshold values are input at execution time, the prototype gives the user the flexibility to try various selection criteria to obtain a reasonable subset of all generated combinations.

With nearly 300 primary features identified in the Ada grammar, it is a very difficult task to identify which of these features are dependent on each other. The approach taken in the ALIANT prototype is to generate hundreds, even thousands, of valid combinations and selectively choose the combinations of interest. The basis for this selection still requires the intuition of the compiler tester since he/she must determine how much duplication is desired, and how many features are "enough". Since it is not practical to test all possible combinations of the language features, the ALIANT selection techniques are based on the premise that combinations being generated repeatedly are more likely to occur in actual use. This premise assumes a "realistically" annotated grammar is used to generate the test combinations. Based on independent tests with the Gen software, it does appear that combinations are being generated according the annotated probabilities. Only by replacing the Gen software with a test case generator customized for ALIANT purposes could total control of the generation process be achieved. Such a "production" version of ALIANT is included as a recommendation for further research.

The testing and analysis presented in the previous chapter showed how various numbers of combinations were tested to illustrate the growth of prototype output. Due to the way the Gen test case generator works, the resulting output is always repeatable. In other words, running 1000 combinations now, and 1000 combinations an hour later, will result in the same set of combinations being generated. Also, if the output from a 1000 combination test run is compared with a 2000 combination test run, the first 1000 combinations of the latter test will match the combinations produced by the former test. Therefore, the only way to get more variety in feature combinations is to run larger and larger test runs.

The development of the ALIANT prototype was made simpler due to several UNIX operating system features. As described in Chapter IV, the use of a UNIX shell script and *redirection* and *piping* features are essential to the operation of the ALIANT prototype. These features reduced the amount of effort required to interface the "off-the-shelf" Gen software, "custom-built" Ada analysis programs, and the Lex support routines. The UNIX environment proved to be ideal for this prototyping effort.

The ALIANT prototype has demonstrated that automated support tools can be used to generate recommended combinations for Ada compiler testing. The utility of the generated combinations still depends on the skill and intuition of the compiler tester, but the prototype makes it possible to analyze large amounts of data in a relatively short time. Once the desired combinations are selected, the database output option allows the information to be used directly by the remaining parts of Ada Features Identification System (AFIS). The ALIANT prototype is usable in its present form, although further improvements may be possible by considering the recommendations that follow.

6.2 Recommendations for Further Research

The completion of the ALIANT prototype confirms the feasibility of automated support tools for generating recommended feature combinations. The optimal implementation of a "production" version of ALIANT would require the entire prototype be developed in a single language, such as Ada. Certain benefits in performance could be realized by eliminating the interfaces between the existing C code (Gen and Lex routines) and the Ada code. Rather than generate thousands of combinations to select from, a production version could be developed to only generate combinations that meet the desired selection criteria. This method should reduce runtime and reduce memory requirements for the intermediate Gen output file. By enhancing the functionality of the Gen portion of the prototype, it may be possible to incorporate additional selection parameters that consider the nesting levels and scope of various feature combinations. Such an implementation would also eliminate the requirement for a UNIX shell script as a user interface. Before a production version is attempted, there are several areas that could benefit from additional prototype enhancement and research.

The first area concerns the operation of the existing ALIANT prototype. During the testing and analysis of the prototype, two potential capabilities were identified. The first would allow the specification of a *starting feature.* Currently, the ALIANT prototype generates a specified number of the *compilation* feature. Since the *compilation* is the highest level feature in the Ada grammar, this produces combinations of some or all of the lower level features. In certain cases, a compiler tester may want to focus on the possible combinations of a lower level feature and generate, say, 1000 combinations of *package_specification.* An enhancement to the ALIANT prototype would allow a user specified "starting point" as another parameter. A default for this parameter could be the *compilation* feature.

The second potential capability concerns the combination matrix. The execution time for generating and tabulating extremely large numbers of combinations can be measured in hours. Depending on the computer loading factor, a test run of 10000 combinations usually takes over an hour. If the user knows ahead of time what threshold values are desired, the ALIANT batch input works very well. On the other hand, if interactive "trial and error" of various threshold options is desired, the prototype cannot be executed in the background with a batch input file. Another enhancement to the existing prototype would be additional options to save/load the combinations matrix to/from memory. This would allow the prototype to generate the combinations matrix as a batch job for later interactive analysis by the user.

The next recommended area for further research is grammar annotation. The grammar annotation is a key step in the ALIANT prototype. Further research into other annotation techniques may yield improvements in the selection/recommendation process. The existing prototype uses duplication and feature counts as selectors. By adding additional embedded indicators in the annotated grammar, it may be possible to allow more complex selection factors to be used. In addition to improvements in the grammar annotation, research in this area may require modification of the Gen software to allow additional annotations to direct the generation process. For example, it may be desirable to have runtime counters to indicate the levels of the feature combinations being generated. This would make it possible to specify a limit on the level, and thereby complexity, of the generated test combinations.

A final recommendation for further research is to investigate the generation of compilable test cases based on the recommended combinations. As it stands right now, the ALIANT prototype identifies combinations of Ada features to be tested. A user must then manually create test cases containing the specified Ada features. Each recommended combination of Ada features can be used to generate many different test cases based on the valid permutations of the Ada features. In other words, the recommended combinations do not specify the context in which the features are used. Further research is needed to develop an automatic technique to generate compilable test cases from the recommended combinations.

6.3 A Final Word

This ALIANT prototype was developed independent of the on-going work on the remaining components of AFIS. Now that the initial feasibility has been demonstrated, the next logical step is to implement the interfaces between ALIANT and the other support tools. The current database output option in ALIANT would have to be adjusted, as necessary, to match the format required for the PAT. Once these minor adjustments have been made, the AMO can begin using the prototype for improving the ACVC test suite. As new grammar specifications are created for the next Ada standard, Ada 9X, the ALIANT prototype will provide the means to identify new feature dependencies and recommend new combinations for testing Ada compilers.

Appendix A. *Gen - A Test Case Generation Program*

This appendix describes the *Gen* test case generation program which was developed by Glenn Kasten, of Ready Systems, California (28). Although it was created to test assembler language programs, it can also be used for other language applications. Examples in this appendix will demonstrate how Gen can be used to produce test cases for Ada compilers.

A.1 Grammars

The input to Gen is a *grammar* which describes the possible *sentences* of a language. The formal grammar for the Ada programming language is described in Appendix E of the Ada Language Reference Manual (LRM) (16). This grammar provides the *syntax* of Ada using a Backus-Naur Form (BNF) format. The Ada grammar uses several special characters (i.e., "::=", "|", "{ }", "[ ]") and typefaces (i.e., boldface and normal) to define valid language constructs. For example, the following Ada *production* defines the proper syntax for an Ada *case_statement*:

```
case_statement ::=
        case expression is
            case_statement_alternative
            {case_statement_alternative}
        end case;
```

In the example above, the boldface words and the semicolon are *terminals* or *terminal symbols*. A terminal symbol will appear in a case statement exactly as shown in the grammar production. The *nonterminals* or *nonterminal symbols*, such as "expression" and "case_statement_alternative", require further expansion by other productions in the Ada grammar. In a *context-free* grammar such as Ada, each production has a single nonterminal on the left-hand side of the symbol "::=". This symbol is equivalent to the phrases: "is defined as", "may be rewritten as", or simply "equals". Wherever the nonterminal on the left-hand side of the equals symbol appears in a grammar production, it can be replaced by the terminals and nonterminals on the right-hand side of the equals symbol. Additional special purpose characters are used to describe the grammar options. These symbols include the vertical bar "|" for alternatives, braces "{ }" to indicate zero or more of the enclosed terminals/nonterminals, and brackets "[ ]" to indicate an optional part of the grammar production.

A.2 Ada Grammar to Gen Grammar

The Gen grammar description constructs are slightly different than the BNF format used in the Ada LRM. For example, the earlier case statement production appears as follows when translated into the Gen input format:

```
case_statement = (
    " case" expression " is"
        case_statement_alternative
        more_alternatives
    " end case;"
)
more_alternatives = (
    " " |
    case_statement_alternative    more_alternatives
)
```

Note that the terminals are enclosed in quotes and the regular equals sign is used instead of the "::=" symbol for each production. In addition to the implied productions that are not shown for "expression" and "case_statement_alternative", a production is required to model the *zero or more* notation from the Ada BNF grammar. The production, *more_alternatives*, allows either the null string or a *case_statement_alternative* followed by *more_alternatives*. In other words, the production *more_alternatives* will generate zero or more occurrences of *case_statement_alternative*.

This simple case_statement example shows how the Gen grammar productions are formed. The following two sections provide the detailed rules for building Gen grammar productions and adding randomness to production alternatives.

A.3 Building Rules

Each Gen grammar production must begin with a nonterminal symbol on the left side of the equals sign. The right side of the equals sign is the rule that describes one or more sentences in the language. The rules can be formed in any combination of the following ways:

- A rule is a nonterminal symbol.
- A rule is a terminal enclosed in double quote marks.
- A rule is two rules separated by a space (concatenation).
- A rule is two rules separated by a vertical bar (alternation).
- A rule is a rule enclosed in parenthesis (grouping). (28:5)

To simplify the structure of rules, Gen provides several short hand notations:

- Alternation of Sets: The rule **vowel = [aeiou]** is equivalent to **vowel = "a" | "e" | "i" | "o" | "u"** and **letter = [a-zA-Z]** is equivalent to listing all the letters individually within brackets.

- Optional Rule: The rule **plural** = ? "s" is the same as **plural** = " " | "s" .
- Integer Range: The rule **octal-byte** = # **0 255 "% 03o"** describes all the octal numbers between 0 and 255, printed with up to two leading zeros. The C language *printf* string in quotes determines the output format of the generated integers. This integer range example is equivalent to the following set notation method:

  octal-byte = [0-3] [0-7] [0-7] (28:6)

In addition to the productions, an executable Gen grammar requires at least one *generation* statement. A generation statement is simply a rule (the right hand side of a production) on a line by itself. If the generation statement is omitted, Gen will not produce any output.

A.4 Randomness Constructs

The percent-sign is used to add randomness to alternation rules. Consider the following production that will generate both A and B:

both = "A" | "B"

By adding the percent-sign after the alternation symbol, a new production is created that will generate either A or B but never both:

either = "A" | % "B"

For alternation sets, the percent-sign is used to randomly select one character or integer from the set. When used with the question-mark operator, the percent-sign causes a random selection of the null string or the given rule. The following grammar would generate a single random 4-letter or 6-letter word:

letter = [a-z] %
word = letter letter letter letter ? (letter letter) %

When the percent-sign is used after the alternate or question-mark operator it may be weighted by an integer constant between 0 and 100. This constant determines the percentage probability that the left side of the alternate operator or the given question-mark rule will be chosen. For example, this version of the *either* production will generate an A, 70 percent of the time and a B, 30 percent of the time:

either = "A" | % 70 "B"

And this version of the *word* production will generate 6-letter words 20 percent of the time and 4-letter words 80 percent of the time:

letter = [a-z] %
word = letter letter letter letter ? (letter letter) % 20

A.5 Using the Test Case Generator

To demonstrate how Gen produces test cases from an input grammar, the previously described case_statement grammar productions are used. The following listing is an executable Gen input grammar for the case_statement example.

```
case_statement = (
    " case" expression " is"
       case_statement_alternative
       more_alternatives
    " end case;"
)

expression = " Exp"

case_statement_alternative = " Case_Alter"

more_alternatives = (
    "" |
    case_statement_alternative  more_alternatives
)

case_statement
```

The only differences between this grammar and the previous Gen case_statement grammar are the addition of the simplified productions for "expression" and "case_statement_alternative" and the occurrence of the "case_statement" *generation statement*. The generation statement tells Gen to generate all the sentences described by the case_statement production. The first few lines of the resulting output from Gen are provided below:

```
case Exp is Case_Alter end case;
case Exp is Case_Alter Case_Alter end case;
case Exp is Case_Alter Case_Alter Case_Alter end case;
case Exp is Case_Alter Case_Alter Case_Alter Case_Alter end case;
case Exp is Case_Alter Case_Alter Case_Alter Case_Alter Case_Alter end case;
```

Since the Gen grammar description for the case_statement can generate an infinite number of sentences, Gen will produce test cases indefinitely. The abbreviated example output above shows that Gen is adding a case_statement_alternative to each sentence to generate the next sentence. To limit the generated sentences to a usable number, Gen has *randomness constructs.* In cases where one or more alternatives are offered, the randomness constructs cause the test case generator to randomly choose between the alternatives rather than exploring all possible alternative combinations. For example, using the case_statement grammar, randomness can be added to the more_alternatives production to limit the number of combinations generated. The modified grammar below includes the "% 60" randomness construct that tells the test case generator to select the null string 60 percent of the time.

```
case_statement = (
    " case" expression " is"
        case_statement_alternative
        more_alternatives
    " end case;"
)

expression = " Exp"

case_statement_alternative = " Case_Alter"

more_alternatives = (
    "" | % 60
    case_statement_alternative  more_alternatives
)

* 15 case_statement
```

The percent-sign operator can drastically reduce the number of alternatives that are chosen. In order to generate enough test cases, the asterisk operator may be used as shown above to repeat the same rule. In this example, 15 case_statement sentences will be generated. Without the asterisk operator, Gen would stop generating test cases as soon as the null string alternative is chosen, resulting in a single test case being generated. The output below shows how the randomness construct changes the types and number of sentences produced:

```
case Exp is Case_Alter end case;
case Exp is Case_Alter end case;
case Exp is Case_Alter end case;
case Exp is Case_Alter end case;
case Exp is Case_Alter end case;
case Exp is Case_Alter Case_Alter end case;
case Exp is Case_Alter end case;
case Exp is Case_Alter Case_Alter end case;
case Exp is Case_Alter end case;
case Exp is Case_Alter end case;
case Exp is Case_Alter end case;
case Exp is Case_Alter end case;
case Exp is Case_Alter Case_Alter end case;
case Exp is Case_Alter Case_Alter Case_Alter end case;
case Exp is Case_Alter end case;
```

Since the internal random generation function within Gen always begins from a fixed "seed" or starting point, the 15 test cases above could not be obtained by 15 independent executions of the given grammar. If this method were used, the output would be 15 copies of the first test case in the listing above. Only by using the asterisk rule can the full benefit of randomness constructs be obtained.

Appendix B. *ACVC Test Class Examples*

The ACVC test suite contains over 4000 legal and illegal Ada test programs divided into six test classes: A, B, C, D, E, and L. Classes A, C, D, and E are executable and use additional utility programs to report test results during execution. The Class B tests are illegal Ada programs that should generate compilation errors. Class L tests should produce compilation or link time errors due to the way the Ada program libraries are used at link time. This appendix provides an example of each test class and a corresponding sample test result from the Verdix Ada compiler. Examples of the utility support programs for reporting executable test results are also provided.

B.1 Report Utility Package

The following package specification shows some of the utility programs used within executable test cases to report test results. Procedure calls are made to these routines to print out the test case being executed and its corresponding pass/fail status.

```
-- REPSPEC.ADA

-- PURPOSE:
--        THIS REPORT PACKAGE PROVIDES THE MECHANISM FOR REPORTING THE
--        PASS/FAIL/NOT-APPLICABLE RESULTS OF EXECUTABLE (CLASSES A, C,
--        D, E, AND L) TESTS.

--        IT ALSO PROVIDES THE MECHANISM FOR GUARANTEEING THAT CERTAIN
--        VALUES BECOME DYNAMIC (NOT KNOWN AT COMPILE-TIME).

-- HISTORY:
--        JRK 12/13/79
--        JRK 06/10/80
--        JRK 08/06/81
--        JRK 10/27/82
--        JRK 06/01/84
--      PWB 07/30/87  ADDED PROCEDURE SPECIAL_ACTION.
--      TBN 08/20/87  ADDED FUNCTION LEGAL_FILE_NAME.

PACKAGE REPORT IS

     SUBTYPE FILE_NUM IS INTEGER RANGE 1..3;

  -- THE REPORT ROUTINES.

     PROCEDURE TEST            -- THIS ROUTINE MUST BE INVOKED AT THE
                               -- START OF A TEST, BEFORE ANY OF THE
                               -- OTHER REPORT ROUTINES ARE INVOKED.
                               -- IT SAVES THE TEST NAME AND OUTPUTS THE
                               -- NAME AND DESCRIPTION.
        ( NAME : STRING;       -- TEST NAME, E.G., "C23001A-AB".
          DESCR : STRING       -- BRIEF DESCRIPTION OF TEST, E.G.,
                               -- "UPPER/LOWER CASE EQUIVALENCE IN " &
                               -- "IDENTIFIERS".
        );

     PROCEDURE FAILED          -- OUTPUT A FAILURE MESSAGE.  SHOULD BE
                               -- INVOKED SEPARATELY TO REPORT THE
                               -- FAILURE OF EACH SUBTEST WITHIN A TEST.
        ( DESCR : STRING       -- BRIEF DESCRIPTION OF WHAT FAILED.
                               -- SHOULD BE PHRASED AS:
                               -- "(FAILED BECAUSE) ...REASON...".
        );

     PROCEDURE NOT_APPLICABLE -- OUTPUT A NOT-APPLICABLE MESSAGE.
                               -- SHOULD BE INVOKED SEPARATELY TO REPORT
                               -- THE NON-APPLICABILITY OF EACH SUBTEST
                               -- WITHIN A TEST.
        ( DESCR : STRING       -- BRIEF DESCRIPTION OF WHAT IS
                               -- NOT-APPLICABLE. SHOULD BE PHRASED AS:
                               -- "(NOT-APPLICABLE BECAUSE)...REASON...".
        );

     PROCEDURE SPECIAL_ACTION -- OUTPUT A MESSAGE DESCRIBING SPECIAL
                               -- ACTIONS TO BE TAKEN.
                               -- SHOULD BE INVOKED SEPARATELY TO GIVE
                               -- EACH SPECIAL ACTION.
        ( DESCR : STRING       -- BRIEF DESCRIPTION OF ACTION TO BE
                               -- TAKEN.
        );

     PROCEDURE COMMENT         -- OUTPUT A COMMENT MESSAGE.
        ( DESCR : STRING       -- THE MESSAGE.
        );

     PROCEDURE RESULT;         -- THIS ROUTINE MUST BE INVOKED AT THE
                               -- END OF A TEST.  IT OUTPUTS A MESSAGE
                               -- INDICATING WHETHER THE TEST AS A
                               -- WHOLE HAS PASSED, FAILED, IS
                               -- NOT-APPLICABLE, OR HAS TENTATIVELY
                               -- PASSED PENDING SPECIAL ACTIONS.
```

```
 -- THE DYNAMIC VALUE ROUTINES.

    -- EVEN WITH STATIC ARGUMENTS, THESE FUNCTIONS WILL HAVE DYNAMIC
    -- RESULTS.

    FUNCTION IDENT_INT        -- AN IDENTITY FUNCTION FOR TYPE INTEGER.
       ( X : INTEGER          -- THE ARGUMENT.
       ) RETURN INTEGER;      -- X.

    FUNCTION IDENT_CHAR       -- AN IDENTITY FUNCTION FOR TYPE
                              -- CHARACTER.
       ( X : CHARACTER        -- THE ARGUMENT.
       ) RETURN CHARACTER;    -- X.

    FUNCTION IDENT_BOOL       -- AN IDENTITY FUNCTION FOR TYPE BOOLEAN.
       ( X : BOOLEAN          -- THE ARGUMENT.
       ) RETURN BOOLEAN;      -- X.

    FUNCTION IDENT_STR        -- AN IDENTITY FUNCTION FOR TYPE STRING.
       ( X : STRING           -- THE ARGUMENT.
       ) RETURN STRING;       -- X.

    FUNCTION EQUAL            -- A RECURSIVE EQUALITY FUNCTION FOR TYPE
                              -- INTEGER.
       ( X, Y : INTEGER       -- THE ARGUMENTS.
       ) RETURN BOOLEAN;      -- X = Y.

-- OTHER UTILITY ROUTINES.

    FUNCTION LEGAL_FILE_NAME -- A FUNCTION TO GENERATE LEGAL EXTERNAL
                              -- FILE NAMES.
       ( X : FILE_NUM := 1;   -- DETERMINES FIRST CHARACTER OF NAME.
         NAM : STRING := ""   -- DETERMINES REST OF NAME.
       ) RETURN STRING;       -- THE GENERATED NAME.

END REPORT;
```

B.2 Class A Test Example

A Class A test is designed to compile successfully. As the following example shows, the execution of the test code will not do anything significant other than call the RESULT report procedure to indicate a successful test.

```
-- A21001A.ADA

-- CHECK THAT THE BASIC CHARACTER SET IS ACCEPTED
-- OUTSIDE OF STRING LITERALS AND COMMENTS.
-- DCB 1/22/80

WITH REPORT;
PROCEDURE A21001A IS

     USE REPORT;

BEGIN
     TEST ("A21001A", "CHECK THAT BASIC CHARACTER SET IS ACCEPTED" );

     DECLARE

          TYPE TABLE IS ARRAY (1..10) OF INTEGER;
          A : TABLE := ( 2 | 4 | 10 => 1 , 1 | 3 | 5..9 => 0 ) ;
                                        -- USE OF : ( ) | ,

          TYPE BUFFER IS
               RECORD
                    LENGTH : INTEGER;
                    POS : INTEGER;
                    IMAGE : INTEGER;
               END RECORD;           -- USED TO TEST . LATER
          R1 : BUFFER;

          ABCDEFGHIJKLM : INTEGER;   -- USE OF A B C D E F G H I J K L M
          NOPQRSTUVWXYZ : INTEGER;   -- USE OF N O P Q R S T U V W X Y Z
          Z_1234567890  : INTEGER;   -- USE OF _ 1 2 3 4 5 6 7 8 9 0

          I1, I2, I3 : INTEGER;
          C1, C2 : STRING (1..6);
          C3 : STRING (1..12);

     BEGIN

          I1 := 2 * ( 3 - 1 + 2 ) / 2 ; I2 := 8 ;  -- USES ( ) * + - / ;

          C1 := "ABCDEF" ;              -- USE OF "
          C2 := C1;

          C1 := "ABCDEF" ;              -- USE OF "
          C2 := C1;
          C3 := C1 & C2 ;               -- USE OF &

          I2 := 16#D#;                  -- USE OF #

          I3 := A'LAST;                 -- USE OF '

          R1.POS := 3;                  -- USE OF .

          IF I1 > 2 AND
             I1 = 4 AND
             I1 < 8 THEN                -- USE OF > = <
               NULL;
          END IF;
```

```
    END;

    RESULT;
END A21001A;
```

The following output shows that the test case compiles successfully:

```
Elxsi Verdix Ada Compiler, Copyright 1984, 1985, 1986, 1987
Version 5.5 - ELXSI UNIX VADS

File: /en0/gcs90d/jmarr/adadir/atest.a
        compiled Mon Jun 18 11:18:59 1990
        by user jmarr

unit:   subprogram body a21001a
        NO Ada ERRORS           UNIT ENTERED

31 statements   64 lines
optimization pass       1       a21001a..NLSB
        102     IL instructions in
        71      IL instructions out
```

When the test case is executed, the following output is produced:

```
---- A21001A CHECK THAT BASIC CHARACTER SET IS ACCEPTED.
==== A21001A PASSED ============================.
```

B.3 Class B Test Example

A Class B test case is designed to produce compilation errors. A Class B test case is passed if every illegal construct is detected at compile time. Note that this test case includes a parameter ($BLANKS) that is used to customize this test case to implementations using fixed length input lines.

```
-- B22001A.TST

-- CHECK THAT AN IDENTIFIER, RESERVED WORD, COMPOUND SYMBOL,
-- INTEGER LITERAL, CHARACTER LITERAL, STRING LITERAL, OR COMMENT
-- CANNOT BE CONTINUED ACROSS A LINE BOUNDARY.

-- FOR IMPLEMENTATIONS THAT USE FIXED LENGTH INPUT LINES,
-- ADDITIONAL BLANKS MUST NOT BE ADDED TO THE END OF THOSE LINES
-- THAT TRY TO FORCE A LEXICAL TOKEN ACROSS A LINE BOUNDARY.
-- THUS, SUFFICIENT (I.E., MAX_IN_LEN - 20) BLANKS ARE MACRO EXPANDED
-- AT THE START OF THOSE PARTICULAR LINES SO AS TO BRING THE
-- LINE LENGTH UP TO THE MAXIMUM ALLOWED INPUT LINE LENGTH.

-- IDENTIFIER CROSSES LINE BOUNDARY.

-- DCB 12/18/79
-- JRK 4/21/80
-- JRK 12/16/80

PROCEDURE B22001A IS

     TYPE INTE IS NEW INTEGER;
     I  : INTEGER;
     EX : INTEGER;
     I1 : INTEGER;
     B1 : BOOLEAN;
     C1 : CHARACTER;
     S1 : STRING (1..6);
$BLANKS         INT1 : INTE
GER;                     -- ERROR: IDENTIFIER CROSSES LINE BOUNDARY.

     I2 : INTEGER;
     I3 : INTEGER;

BEGIN

     NULL;

     WHILE FALSE LOOP
          NULL;
     END LOOP;

END B22001A;
```

The following output shows that the compiler does detect the intended error:

```
Elxsi Verdix Ada Compiler, Copyright 1984, 1985, 1986, 1987
Version 5.5 - ELXSI UNIX VADS

File: /en0/gcs90d/jmarr/adadir/btest.a
        compiled Mon Jun 18 11:20:00 1990
        by user jmarr

unit:   subprogram subunit b22001a
        1 SYNTAX ERROR          UNIT UNCHANGED

14 statements   43 lines

**********************************  btest.a  **********************************

  30:GER;                -- ERROR: IDENTIFIER CROSSES LINE BOUNDARY.
A ---^
A:syntax error: "ger" deleted
```

B.4 Class C Test Example

A Class C test case is designed to check the run time system. These test cases include code that must compile and execute successfully. If the code does not execute as expected, a procedure call is made to the utility program FAILED to print out an error message.

```
-- C23001A.ADA

-- CHECK THAT UPPER AND LOWER CASE LETTERS ARE EQUIVALENT IN IDENTIFIERS
-- (INCLUDING RESERVED WORDS).

-- JRK 12/12/79
-- JWC 6/28/85   RENAMED TO -AB

WITH REPORT;
PROCEDURE C23001A IS

     USE REPORT;

     AN_IDENTIFIER : INTEGER := 1;

BEGIN
     TEST ("C23001A", "UPPER/LOWER CASE EQUIVALENCE IN IDENTIFIERS");

     DECLARE
             an_identifier : INTEGER := 3;
     BEGIN
             IF an_identifier /= AN_IDENTIFIER THEN
                     FAILED ("LOWER CASE NOT EQUIVALENT TO UPPER " &
                             "IN DECLARABLE IDENTIFIERS");
             END IF;
     END;
     IF An_IdEnTIfieR /= AN_IDENTIFIER THEN
             FAILED ("MIXED CASE NOT EQUIVALENT TO UPPER IN " &
                     "DECLARABLE IDENTIFIERS");
     END IF;

     if AN_IDENTIFIER = 1 ThEn
             AN_IDENTIFIER := 2;
     END IF;
     IF AN_IDENTIFIER /= 2 THEN
             FAILED ("LOWER AND/OR MIXED CASE NOT EQUIVALENT TO " &
                     "UPPER IN RESERVED WORDS");
     END IF;

     RESULT;
END C23001A;
```

The following output shows that the Class C test compiles successfully:

```
Elxsi Verdix Ada Compiler, Copyright 1984, 1985, 1986, 1987
Version 5.5 - ELXSI UNIX VADS

File: /en0/gcs90d/jmarr/adadir/ctest.a
        compiled Mon Jun 18 11:20:12 1990
        by user jmarr

unit:   subprogram body c23001a
        NO Ada ERRORS           UNIT ENTERED

16 statements   43 lines

**********************************  ctest.a  *********************************

  20:            an_identifier : INTEGER := 3;
A ----------------^
A:warning: id hides outer definition
optimization pass       1       c23001a..NLSB
        78       IL instructions in
        62       IL instructions out
```

When the test case is executed, the following output is produced:

```
---- C23001A UPPER/LOWER CASE EQUIVALENCE IN IDENTIFIERS.
==== C23001A PASSED ============================.
```

B.5 Class D Test Example

A Class D test case checks the compilation and execution capacities of a compiler. Since most capacity limits are not specified by the Ada language standard, a valid compiler may be classified as *inapplicable* if a Class D test fails to compile because the capacity of the compiler is exceeded.

```
-- D55A03A.ADA

-- CHECK THAT AN ARBITRARY LEVEL OF LOOP NESTING IS PERMITTED.

-- CHECK 7 LEVELS OF LOOP NESTING.

-- ASL 8/06/81
-- RM 6/28/82
-- RM 7/06/82
-- SPS 3/1/83

WITH REPORT;
PROCEDURE D55A03A IS

     USE REPORT;

     X : INTEGER := 1;

     COUNT : INTEGER := 0;

     DESCENDING  :  BOOLEAN  :=  IDENT_BOOL( TRUE );

BEGIN

     TEST ("D55A03A","7 LEVELS OF LOOP NESTING");

     FOR I IN X..IDENT_INT(1) LOOP
     WHILE  DESCENDING  LOOP
     LOOP
     EXIT WHEN  NOT DESCENDING ;

     FOR I IN X..IDENT_INT(1) LOOP
     WHILE  DESCENDING  LOOP
     LOOP
     EXIT WHEN  NOT DESCENDING ;

     FOR I IN X..IDENT_INT(1) LOOP

          COUNT := COUNT + 1;
          DESCENDING  :=  IDENT_BOOL( FALSE );
      END LOOP;
      END LOOP;
      END LOOP;
      END LOOP;
      END LOOP;
      END LOOP;
      END LOOP;

      IF COUNT /= 1 THEN
           FAILED ("LOOPS NOT EXECUTED PROPER NUMBER OF TIMES");
      END IF;

      RESULT;

END D55A03A;
```

The following output shows the results of a successful compilation of this Class D test:

```
Elxsi Verdix Ada Compiler, Copyright 1984, 1985, 1986, 1987
Version 5.5 - ELXSI UNIX VADS

File: /en0/gcs90d/jmarr/adadir/dtest.a
        compiled Mon Jun 18 11:25:03 1990
        by user jmarr

unit:   subprogram body d55a03a
        NO Ada ERRORS           UNIT ENTERED

21 statements   56 lines

********************************** dtest.a **********************************

  33:     FOR I IN X..IDENT_INT(1) LOOP
A ------------^
A:warning: id hides outer definition
  38:     FOR I IN X..IDENT_INT(1) LOOP
A ------------^
A:warning: id hides outer definition
optimization pass       1       d55a03a..NLSB
        176      IL instructions in
        132      IL instructions out
```

When the test is executed, the following output is produced:

```
---- D55A03A 7 LEVELS OF LOOP NESTING.
==== D55A03A PASSED ==========================.
```

B.6 Class E Test Example

A Class E test is designed to check implementation-dependent options. Like a Class D test, a Class E test may be inapplicable to a certain compiler implementation.

```
PRAGMA SYSTEM_NAME (NOBODY);
PRAGMA MEMORY_SIZE (ONE);
PRAGMA STORAGE_UNIT (TWO);

-- E28002A.ADA

-- OBJECTIVE:
--     CHECK THAT A PREDEFINED OR AN UNRECOGNIZED PRAGMA MAY HAVE
--     ARGUMENTS INVOLVING IDENTIFIERS THAT ARE NOT VISIBLE.

--     THESE PRAGMAS ARE IMPROPER, BUT THEY ARE LEGAL STATEMENTS
--     THAT MUST BE IGNORED BY THE COMPILER.

-- PASS/FAIL CRITERIA:
--     1) THE TEST MUST EXECUTE AND REPORT "TENTATIVELY PASSED";
--     2) THE COMMENT CONTAINING "*** MUST APPEAR ***" MUST APPEAR IN
--        THE COMPILATION LISTING;
--     3) THE TWO COMMENTS CONTAINING "*** SAME PAGE ***" MUST APPEAR ON
--        THE SAME PAGE.

-- HISTORY:
--     TBN 02/21/86  CREATED ORIGINAL TEST.
--     JET 01/13/88  ADDED CALLS TO SPEC_ACT AND UPDATED HEADER FORMAT.
--     DHH 03/02/89  ADDED PRAGMA PAGE BEFORE PRAGMA PAGE(ONE).

WITH REPORT; USE REPORT;
PRAGMA ELABORATE (ZZZZZZZ_ZZZ);

PROCEDURE E28002A IS

     PRAGMA OPTIMIZE (WHAT);
     PRAGMA PRIORITY (ONE);
     PRAGMA CONTROLLED (OPTIMIZE);
     PRAGMA SHARED (GLOBAL_MONEY);
     PRAGMA INTERFACE (FORTRAN, FUN);
     PRAGMA INLINE (XYZ);
     PRAGMA PACK (CHAR_TYPE);
     PRAGMA SUPPRESS (MONEY, INTEGER);
     PRAGMA PHIL_BRASHEAR (ONE);
     MY_INT : INTEGER;

BEGIN
     TEST ("E28002A", "CHECK THAT A PREDEFINED OR AN UNRECOGNIZED " &
           "PRAGMA MAY HAVE ARGUMENTS INVOLVING " &
           "IDENTIFIERS THAT ARE NOT VISIBLE");
     PRAGMA LIST (NEXT);
-- THIS COMMENT *** MUST APPEAR ***.
     SPECIAL_ACTION ("CHECK LISTING FOR COMMENT ""*** MUST APPEAR " &
                     "***""");
     PRAGMA ROSA_WILLIAMS (TWO);
     PRAGMA THOMAS_NORRIS (THREE);
     PRAGMA PAGE;
     PRAGMA PAGE (ONE);
-- THIS COMMENT MUST BE ON THE *** SAME PAGE *** AS THE NEXT COMMENT.
     PRAGMA PAGE (FOUR);
-- THIS COMMENT MUST BE ON THE *** SAME PAGE *** AS THE PRECEDING
-- COMMENT.
     SPECIAL_ACTION ("CHECK THAT COMMENTS ""*** SAME PAGE ***"" " &
                     "ARE ON THE SAME PAGE OF THE LISTING");
     RESULT;

END E28002A;
```

The following output shows that the Class E test does compile successfully:

(Although several warnings are generated)

```
Elxsi Verdix Ada Compiler, Copyright 1984, 1985, 1986, 1987
Version 5.5 - ELXSI UNIX VADS

File: /en0/gcs90d/jmarr/adadir/etest.a
        compiled Mon Jun 18 11:29:10 1990
        by user jmarr

unit:   subprogram body e28002a
        NO Ada ERRORS          UNIT ENTERED

27 statements   62 lines

*********************************  etest.a  *********************************

   1:PRAGMA SYSTEM_NAME (NOBODY);
A ---^
A:warning: RM 13.7(11) System can be altered only by direct recompilation
   2:PRAGMA MEMORY_SIZE (ONE);
A ---^
```

```
B ----------------------^
A:warning: RM 13.7(11) System can be altered only by direct recompilation
B:warning: RM Appendix B: pragma argument must be a numeric literal
   3:PRAGMA STORAGE_UNIT (TWO);
A ---^
B ----------------------^
A:warning: RM 13.7(11) System can be altered only by direct recompilation
B:warning: RM Appendix B: pragma argument must be a numeric literal
  27:PRAGMA ELABORATE (ZZZZZZZ_ZZZ);
A --------------------^
A:warning: RM B(3): doesn't name WITH'd unit.
  31:     PRAGMA OPTIMIZE (WHAT);
A ------------------------^
A:warning: RM Appendix B: incorrect pragma argument identifier
  32:     PRAGMA PRIORITY (ONE);
A --------^
A:warning: RM 13.7: SYSTEM is not available
  33:     PRAGMA CONTROLLED (OPTIMIZE);
A ---------------------------^
A:warning: RM 8.3: identifier undefined
  34:     PRAGMA SHARED (GLOBAL_MONEY);
A -----------------------^
A:warning: RM 8.3: identifier undefined
A:warning: RM 9.11(10): argument must be a variable
  35:     PRAGMA INTERFACE (FORTRAN, FUN);
A ----------------------------------^
A:warning: RM 8.3: identifier undefined
A:warning: RM 13.9: not a subprogram name
  36:     PRAGMA INLINE (XYZ);
A ----------------------^
A:warning: RM 6.3.2(3): must name a subprogram in the current declarative part
  37:     PRAGMA PACK (CHAR_TYPE);
A ---------------------^
A:warning: RM 8.3: identifier undefined
  38:     PRAGMA SUPPRESS (MONEY, INTEGER);
A -------------------------^
A:warning: RM Appendix B: incorrect pragma argument identifier
  39:     PRAGMA PHIL_BRASHEAR (ONE);
A --------^
A:warning: RM Appendix B: undefined pragma
  46:     PRAGMA LIST (NEXT);
A ---------------------^
A:warning: RM Appendix B: incorrect pragma argument identifier
  50:     PRAGMA ROSA_WILLIAMS (TWO);
A --------^
A:warning: RM Appendix B: undefined pragma
  51:     PRAGMA THOMAS_NORRIS (THREE);
A --------^
A:warning: RM Appendix B: undefined pragma
  53:     PRAGMA PAGE (ONE);
A --------------------^
A:warning: RM Appendix B: too many pragma arguments
  55:     PRAGMA PAGE (FOUR);
A --------------------^
A:warning: RM Appendix B: too many pragma arguments
optimization pass       1       e28002a..NLSB
```

```
47        IL instructions in
32        IL instructions out
```

When the test is executed, the following output is produced. As indicated by the output comments, the test is not completely passed until the source code listing is checked for proper location of the test comments. Although not included here, the source listing did, in fact, have the test comments on the proper pages.

```
---- E28002A CHECK THAT A PREDEFINED OR AN UNRECOGNIZED PRAGMA MAY HAVE
                ARGUMENTS INVOLVING IDENTIFIERS THAT ARE NOT VISIBLE.
   ! E28002A CHECK LISTING FOR COMMENT "*** MUST APPEAR ***".
   ! E28002A CHECK THAT COMMENTS "*** SAME PAGE ***" ARE ON THE SAME
                PAGE OF THE LISTING.
!!!! E28002A TENTATIVELY PASSED !!!!!!!!!!!!!!!!.
!!!!         SEE '!' COMMENTS FOR SPECIAL NOTES!!
```

B.7 Class L Test Example

A Class L test case is designed to fail no later than link time (some implementations may detect an error at compile time). In most cases, these tests check that "incomplete or illegal Ada programs involving multiple, separately compiled units are detected and not allowed to execute" (2:1.5).

```
-- LA1001F.ADA

-- OBJECTIVE:
--      CHECK THAT A PACKAGE CANNOT BE NAMED AS A MAIN PROGRAM.

-- HISTORY:
--      JET 08/12/88  CREATED ORIGINAL TEST.

PACKAGE LA1001F IS
END LA1001F;

WITH REPORT; USE REPORT;
PRAGMA ELABORATE(REPORT);
```

```
PACKAGE BODY LA1001F IS
BEGIN
     TEST("LA1001F", "CHECK THAT A PACKAGE CANNOT BE NAMED AS A " &
                     "MAIN PROGRAM");
     FAILED("PACKAGE WAS IMPROPERLY LINKED AND EXECUTED");

     RESULT;
END LA1001F;
```

The following output shows that the Class L test compiled successfully:

```
Elxsi Verdix Ada Compiler, Copyright 1984, 1985, 1986, 1987
Version 5.5 - ELXSI UNIX VADS

File: /en0/gcs90d/jmarr/adadir/ltest.a
        compiled Mon Jun 18 11:31:25 1990
        by user jmarr

unit:   package la1001f
        NO Ada ERRORS           UNIT ENTERED
unit:   package body la1001f
        NO Ada ERRORS           UNIT ENTERED

8 statements    21 lines
optimization pass       1       la1001f..NLPS
        9        IL instructions in
        7        IL instructions out
optimization pass       1       la1001f..NLPB
        36       IL instructions in
        26       IL instructions out
```

When the test is linked, the following error message is produced, as desired:

```
RM 10.1(8): spec of la1001f (from /en0/gcs90d/jmarr/adadir/ltest.a)
        can not be a main program
        A parameterless integer function or procedure is required
```

Appendix C. *Lex Description*

The ALIANT prototype depends on a utility program called Lex. Lex is used to create a C language subroutine which recognizes character strings produced by the Gen software. This appendix describes the operation of Lex and some of the ways it can be used to generate stand-alone programs or subroutines. Sample Lex specification files used by the ALIANT prototype are also provided.

C.1 Lex Description

Lex is a generator of lexical analysis programs. A lexical analyzer is the first phase of a compiler. "The function of the lexical analyzer is to read the source program, one character at a time, and to translate it into a sequence of primitive units called tokens. Keywords, identifiers, constants, and operators are examples of tokens" (4:73). These tokens are then passed to the next phase, the syntax analyzer, or parser.

Lex simplifies the somewhat tedious task of creating a lexical analyzer. Given an input file containing properly formatted regular expressions and associated actions, Lex will automatically generate a C language program. When compiled and executed, this program will search for the user specified regular expressions in an input text file. If one of the regular expressions is found, the associated action is performed. Lex was developed by M. E. Lesk and E. Schmidt at Bell Laboratories in 1975 (30).

Lex programs are usually created to provide input to a parser generated by YACC. YACC is an automatic generator for the parser phase of a compiler. It was designed by S. C. Johnson and presented is his paper, *YACC - Yet Another Compiler Compiler* (26).

> YACC produces a parser which continually invokes a user defined lexical analyzer to process streams of input. The specification of the grammar includes a list of tokens for the grammar, the grammar rules and any actions to be taken as the rules are invoked. The actions have the ability to return values and to use the values returned by other actions. (43:23)

When used in this manner, the Lex program is "included" in the YACC specification file, allowing YACC to make function calls to the Lex subroutine. The YACC specification file, which will not be discussed in more detail, consists of three sections: the declaration section, the grammar rules section and the program section. A thorough discussion of how Lex and YACC work together can be found in a thesis by Rosa J. Williams, *Automatic Generation of Parsers Using YACC and Lex* (43).

Lex programs can also be used independently from YACC as a "stand-alone" program or as a subroutine to other user defined programs. This allows the user to take full advantage of the powerful recognition capability of regular expressions to search various sorts of input text files.

A Lex program is defined by a *specification* file. The Lex specification file consists of definitions, rules, and user subroutines. The format is as follows:

```
definitions
%%
rules
%%
user subroutines
```

The definitions and user subroutines are optional and may be omitted. The smallest Lex specification file is just "%%"; in which the implied rule is to copy the input file to the output file unchanged. When rules are specified, they have the general form "expression   action". The expression is a regular expression that will be described in the next paragraph. The actions are user defined C language statements. Whenever the regular expression is recognized in the input text, the corresponding action, if any, is executed. Any portion of the input file that does not match a regular expression is copied to the output file. The Lex specification file can be created using any text editor and stored in a user file for Lex processing. When Lex is invoked with the name of the specification file, the generated C program will be stored in "lex.yy.c". Further details for processing the specification file are provided in a later paragraph.

The regular expressions used by Lex are similar to those used by various UNIX pattern recognition programs such as "awk" and "grep". Regular expressions may contain letters, digits and operator symbols. The following special characters are considered operator symbols by Lex:

```
" \ [ ] ^ - ? . * + | ( ) $ / { } % < >
```

If any of these characters is to be used as a text character, it must be preceded by the escape character, \ (backslash), or be included within quotation marks. For example, each of the following regular expressions will recognize the string "count++":

```
"count++"     count"++"      count\+\+
```

The left and right brackets, [and], are used to denote character classes. The character class [aAbBcC] will match a single upper or lowercase A, B, or C. By using the brackets in conjunction with the operator symbols \, ^, and -; a variety of recognition patterns can be created. A common example is [a-z0-9], which will match a single lowercase letter or digit. To match any character besides a-z or 0-9, the ^ operator symbol is included: [^a-z0-9]. But to match a-z, 0-9, and ^, the escape symbol is included: [\^a-z0-9]. The backslash tells Lex to treat the caret symbol as a text character rather than a control symbol.

The ? is used to denote optional characters. The expression st?k will match sk or stk. By adding the repetition operator symbols * and + , more complicated text strings can be recognized. The * symbol indicates *zero or more* occurrences of a text character or string, and the + symbol indicates *one or more* occurrence. For example, st+?k* would match such strings as s, st, stk, sttkk, and sk; but not k. The expression [a-z]+ will recognize all strings of one or more lowercase letters. The expression [A-Za-z][A-Za-z0-9]* will match all alphanumeric strings beginning with a letter (43:46).

The operator symbol | denotes alternation while the parentheses are used to group complex expressions. The expression (abc|xyz) will match either abc or xyz. The expression (abc+|xyz*) will match such strings as abc, abcc, xy, xyz, xyzz.

> A more practical example is [a-zA-Z]([_]?[a-zA-Z0-9])*. This expression matches any identifier for the Ada language. Note that the identifier must begin with a letter and it may contain zero or more additional letters or numbers. The "_" character is optional, but should it occur, it must be followed by at least one letter or number. Hence, consecutive underscores or terminating underscores are not allowed. (43:47)

Some of the operator symbols can be used to indicate the context in which a regular expression is to be recognized. For example, the string ^[a-z] will match any lowercase character if it is located at the beginning of a line, whereas [a-z]$ will match the same character if it occurs at the end of a line.

The remaining operator symbols are used in conjunction with the definition section of the specification file. A simple example of the definition section is provided below. Further information can be found in (30) or (43). To simplify some regular expressions, or to make them more readable, the definition section can be used to give names to specific expressions. For example, consider the following specification file:

```
e      [eE]                                         }
digit  [0-9]                                        }definition
digits {digit}([_]?{digits})*                       }section
%%
{digits} ({e}[+]?{digits})?   printf("integer"); }rules section
```

The braces, { and }, denote repetition if they enclose numbers, or definition expansion if they enclose a name. In the example above, the braces are used for definition expansion, but in the string a{1,5} the braces mean one to five occurrences of the letter "a". The regular expression in the rules section above will match an integer literal from the Ada language and the action will cause "integer" to be printed (43). The next paragraph shows how an actual Lex specification file is processed to create the Lex program.

To illustrate the steps in creating a Lex program, the following simple Lex specification is borrowed from the UNIX User's Manual (38):

```
%%
[A-Z]  putchar(yytext[0] + 'a' - 'A');
[ ]+$  ;
[ ]+   putchar (' ');
%%
main()
{ yylex(); }
```

First note that this specification file has no declaration section. The rules section has three regular expressions. The first one will match any uppercase letter. The corresponding action will output the letter in lowercase. The array "yytext" is a standard character found in every Lex generated program. It contains the input string matched by a regular expression. In this case, a single character will be located in yytext[0]. By subtracting the ASCII difference between "a" and "A", the resulting character will be the lowercase equivalent of the letter in yytext[0]. The second regular expression/action will strip trailing blanks from each input line, while the third regular expression/action will replace strings of one or more non-trailing blanks with a single blank. The user subroutine section contains a driver program for the Lex program which has the standard name "yylex". This driver will allow the Lex program to execute without interfacing with any other programs. This is a good way to test individual Lex programs before interfacing them to YACC or other routines.

The first step to process the Lex specification file is to invoke Lex using the following command:

```
lex filename
```

where "filename" is the filename of the Lex specification file. The resulting output is automatically stored in "lex.yy.c". The next step in creating an executable Lex program, is to compile the lex.yy.c source code file:

```
cc lex.yy.c -ll
```

The executable file will be stored in a file called "a.out". Figure C.1 shows input/output examples for this program (with comments added). The way this sample program was set up, the keyboard is the default input device and the terminal screen is the default output device. Although this is a simple example, the sample input/output demonstrates that the specified Lex program works properly.

```
%a.out      <--- execute compiled lex.yy.c program

THIS LINE IS IN ALL CAPITAL LETTERS.          <-- input
this line is in all capital letters.          <-- output

this line is MIXED UPPER AND lower case.      <-- input
this line is mixed upper and lower case.      <-- output

THIS LINE HAS EXTRA     SPACES IN THE MIDDLE. <-- input
this line has extra spaces in the middle.     <-- output
```

Figure C.1. Sample Yylex Input/Output

As Lex processes a specification file, it will identify any errors found in rules syntax; however, the C code in the action statements will not be checked for errors until the lex.yy.c file is processed by the C compiler. Depending on the size of the specification file and complexity of the regular expressions, the default sizes for the Lex generated tables may be exceeded. If so, Lex will display an error message indicating the name of the table that overflowed and the current size limit of the table. To increase the size of any of the tables, a statement must be added in the declarations section of the specification file. The format is "%x nnn" where nnn is a decimal integer representing the table size and x is one of the parameters listed in Table C.1.

Table C.1. Lex Size Parameters

Letter	Parameter
a	transitions
e	tree nodes
k	packed character class
n	states
o	output array size
p	positions

The C source code produced by Lex can be interfaced with other C programs by simply including the lex.yy.c with the other routines before compilation. Interface with Ada programs is also possible by using *pragma interface*, as done in the ALIANT prototype (Chapter IV). The brief Lex sample just presented did not return values to the calling program when it matched a particular regular expression. The typical use of a Lex program is to find a token and return a value for a parser to analyze syntax. That is the way Lex is used in the ALIANT prototype.

C.2 Sample lex_spec Listing

The following listing is the lex_spec file generated from the Adagen2 grammar. For each regular expression, a unique token number is returned. The ALIANT_Driver uses this token to determine what action to take.

```
%a 6000
%e 7000
%n 4000
%p 24000
```

```
%o 5000
%%
graphic_character                       { return(  1); }
basic_graphic_character                 { return(  2); }
basic_character                         { return(  3); }
identifier                              { return(  4); }
letter_or_digit                         { return(  5); }
letter                                  { return(  6); }
integer_literal                         { return(  7); }
real_literal                            { return(  8); }
integer                                 { return(  9); }
exponent                                { return( 10); }
based_literal                           { return( 11); }
base                                    { return( 12); }
based_integer                           { return( 13); }
extended_digit                          { return( 14); }
character_literal                       { return( 15); }
string_literal                          { return( 16); }
pragma                                  { return( 17); }
pragma:argument_association             { return( 18); }
predef_pragma                           { return( 19); }
argument_association                    { return( 20); }
object_decl                             { return( 21); }
object_init_val                         { return( 22); }
object_init_val_constrained_array       { return( 23); }
constant_decl                           { return( 24); }
number_decl                             { return( 25); }
identifier_list                         { return( 26); }
full_type_decl                          { return( 27); }
subtype_decl                            { return( 28); }
subtype_indic                           { return( 29); }
type_mark                               { return( 30); }
derived_type_def                        { return( 31); }
range_attribute                         { return( 32); }
explicit_range                          { return( 33); }
enum_type_def                           { return( 34); }
enumeration_literal_specification       { return( 35); }
enumeration_literal                     { return( 36); }
integer_type_def                        { return( 37); }
floating_point_type_def                 { return( 38); }
fixed_point_type_def                    { return( 39); }
floating_point_constraint               { return( 40); }
floating_accuracy_definition            { return( 41); }
fixed_point_constraint                  { return( 42); }
fixed_accuracy_definition               { return( 43); }
```

```
array_type_def                        { return( 44); }
array_of:access                       { return( 45); }
array_of:boolean                      { return( 46); }
array_of:integer                      { return( 47); }
array_of:real                         { return( 48); }
array_of:record                       { return( 49); }
array_of:task                         { return( 50); }
unconstrained_array_def               { return( 51); }
constrained_array_def                 { return( 52); }
index_subtype_definition              { return( 53); }
index_constraint                      { return( 54); }
discrete_range                        { return( 55); }
record_type_def                       { return( 56); }
record_of:access                      { return( 57); }
record_of:array                       { return( 58); }
record_of:record                      { return( 59); }
record_of:task                        { return( 60); }
null_component_list                   { return( 61); }
component_decl:default                { return( 62); }
component_decl:no_default             { return( 63); }
component_subtype_definition          { return( 64); }
discriminant_spec:default             { return( 65); }
discriminant_spec:no_default          { return( 66); }
discriminant_constraint               { return( 67); }
discriminant_association              { return( 68); }
variant_part                          { return( 69); }
variant_choice                        { return( 70); }
variant_choice_others                 { return( 71); }
access_type_def                       { return( 72); }
access_to:array                       { return( 73); }
access_to:record                      { return( 74); }
access_to:task                        { return( 75); }
incomplete_type_decl                  { return( 76); }
indexed_component                     { return( 77); }
slice                                 { return( 78); }
selected_component                    { return( 79); }
selector_all                          { return( 80); }
attribute                             { return( 81); }
predef_attr                           { return( 82); }
attribute_designator                  { return( 83); }
aggregate                             { return( 84); }
named_component_association           { return( 85); }
andthen                               { return( 86); }
orelse                                { return( 87); }
membership_test_in                    { return( 88); }
```

```
membership_test_not_in                    { return( 89); }
simple_expression                         { return( 90); }
exponentiation                            { return( 91); }
absolute_value                            { return( 92); }
not_operator                              { return( 93); }
null_access_value                         { return( 94); }
parenthesized_expr                        { return( 95); }
and_operator                              { return( 96); }
or_operator                               { return( 97); }
xor_operator                              { return( 98); }
equality                                  { return( 99); }
inequality                                { return(100); }
less_than                                 { return(101); }
less_than_or_equal_to                     { return(102); }
greater_than                              { return(103); }
greater_than_or_equal_to                  { return(104); }
addition                                  { return(105); }
subtraction                               { return(106); }
catenation                                { return(107); }
unary_addition                            { return(108); }
unary_minus                               { return(109); }
multiplication                            { return(110); }
division                                  { return(111); }
mod_operator                              { return(112); }
rem_operator                              { return(113); }
exponentiation                            { return(114); }
absolute_value                            { return(115); }
not_operator                              { return(116); }
type_conversion                           { return(117); }
qualified_expr                            { return(118); }
alloc:qualified_expr                      { return(119); }
alloc:subtype_indic_constr                { return(120); }
alloc:subtype_indic_no_constr             { return(121); }
label                                     { return(122); }
null_statement                            { return(123); }
assignment_statement                      { return(124); }
if_statement                              { return(125); }
condition                                 { return(126); }
case_statement                            { return(127); }
case_statement_alternative                { return(128); }
loop_statement                            { return(129); }
iteration_scheme:for                      { return(130); }
iteration_scheme:while                    { return(131); }
loop_param_spec:up                        { return(132); }
loop_param_spec:down                      { return(133); }
```

```
block_statement                         { return(134); }
exit_statement                          { return(135); }
return_statement                        { return(136); }
goto_statement                          { return(137); }
subprogram_decl:procedure               { return(138); }
subprogram_decl:function                { return(139); }
user_defined_operator                   { return(140); }
subprog_param_spec:default              { return(141); }
subprog_param_spec:in                   { return(142); }
subprog_param_spec:in_default           { return(143); }
subprog_param_spec:in_out               { return(144); }
subprog_param_spec:no_default           { return(145); }
subprog_param_spec:out                  { return(146); }
mode_in                                 { return(147); }
mode_in_default                         { return(148); }
mode_in_out                             { return(149); }
mode_out                                { return(150); }
procedure_body                          { return(151); }
function_body                           { return(152); }
procedure_call_statement                { return(153); }
function_call                           { return(154); }
actual_parameter_part                   { return(155); }
parameter_association                   { return(156); }
formal_parameter                        { return(157); }
actual_parameter                        { return(158); }
package_spec                            { return(159); }
package_body                            { return(160); }
private_type_decl                       { return(161); }
limited_private_type_decl               { return(162); }
deferred_constant_declaration           { return(163); }
use_clause                              { return(164); }
rename:entry                            { return(165); }
rename:exception                        { return(166); }
rename:object                           { return(167); }
rename:package                          { return(168); }
rename:subprog                          { return(169); }
rename:subprog_or_entry                 { return(170); }
task_spec                               { return(171); }
task_type_spec                          { return(172); }
task_body                               { return(173); }
entry_decl                              { return(174); }
entry_family_decl                       { return(175); }
entry_param_spec                        { return(176); }
entry_param_spec:default                { return(177); }
entry_param_spec:in                     { return(178); }
```

```
entry_param_spec:in_default             { return(179); }
entry_param_spec:in_out                 { return(180); }
entry_param_spec:no_default             { return(181); }
entry_param_spec:out                    { return(182); }
entry_call_statement                    { return(183); }
accept_statement                        { return(184); }
delay_statement                         { return(185); }
sel_wait:accept_alt                     { return(186); }
sel_wait:accept_alt_guarded             { return(187); }
sel_wait:accept_alt_unguarded           { return(188); }
sel_wait:delay_alt                      { return(189); }
sel_wait:delay_alt_guarded              { return(190); }
sel_wait:delay_alt_unguarded            { return(191); }
sel_wait:else_part                      { return(192); }
sel_wait:term_alt                       { return(193); }
sel_wait:term_alt_guarded               { return(194); }
sel_wait:term_alt_unguarded             { return(195); }
select_alternative                      { return(196); }
terminate_alternative                   { return(197); }
conditional_entry_call                  { return(198); }
timed_entry_call                        { return(199); }
abort_statement                         { return(200); }
with_clause                             { return(201); }
procedure_body_stub                     { return(202); }
function_body_stub                      { return(203); }
package_body_stub                       { return(204); }
task_body_stub                          { return(205); }
procedure_subunit                       { return(206); }
function_subunit                        { return(207); }
package_subunit                         { return(208); }
task_subunit                            { return(209); }
exception_decl                          { return(210); }
exception_handler                       { return(211); }
exception_choice_others                 { return(212); }
predef_except                           { return(213); }
raise_statement                         { return(214); }
gen_package_spec                        { return(215); }
gen_subprog_spec                        { return(216); }
gen_subprog_spec:function               { return(217); }
gen_subprog_spec:procedure              { return(218); }
gen_formal_obj:default                  { return(219); }
gen_formal_obj:in                       { return(220); }
gen_formal_obj:in_default               { return(221); }
gen_formal_obj:in_out                   { return(222); }
gen_formal_obj:no_default               { return(223); }
```

```
gen_formal_part                         { return(224); }
gen_formal_subprog                      { return(225); }
gen_formal_subprog:box_default          { return(226); }
gen_formal_subprog:nm_default           { return(227); }
gen_formal_type                         { return(228); }
gen_formal_type:access                  { return(229); }
gen_formal_type:array                   { return(230); }
gen_formal_type:discrete                { return(231); }
gen_formal_type:fixed_point             { return(232); }
gen_formal_type:floating_point          { return(233); }
gen_formal_type:integer                 { return(234); }
gen_formal_type:lim_private             { return(235); }
gen_formal_type:private                 { return(236); }
generic_type_definition                 { return(237); }
gen_function_instantiation              { return(238); }
gen_package_instantiation               { return(239); }
gen_procedure_instantiation             { return(240); }
gen_subprog_instantiation               { return(241); }
gen_actual_object                       { return(242); }
gen_actual:subprog                      { return(243); }
gen_actual:type                         { return(244); }
gen_actual:type_access                  { return(245); }
gen_actual:type_array                   { return(246); }
gen_actual:type_discrete                { return(247); }
gen_actual:type_fixed_point             { return(248); }
gen_actual:type_floating_point          { return(249); }
gen_actual:type_integer                 { return(250); }
generic_association                     { return(251); }
generic_formal_parameter                { return(252); }
generic_actual_parameter                { return(253); }
length_clause                           { return(254); }
length_clause:size                      { return(255); }
length_clause:small                     { return(256); }
length_clause:strng_size                { return(257); }
length_clause:strg_size_access          { return(258); }
length_clause:strg_size_access          { return(259); }
length_clause:strg_size_task            { return(260); }
enum_repr_clause                        { return(261); }
record_repr_clause                      { return(262); }
alignment_clause                        { return(263); }
component_clause                        { return(264); }
address_clause                          { return(265); }
code_statement                          { return(266); }
START_COMPILATION:                      { return(995); }
:END_COMPILATION                        { return(996); }
```

```
[a-z\:\_]+                                { return(997); }
[ ]+                                      { return(998); }
[\n]                                      { return(999); }
%%
```

C.3 Sample mk_lspec Listing

The following listing is the Lex specification and driver routine that is used to generate the lex_spec file from the input grammar. The regular expressions are designed to recognize every character expected in the grammar file. Token 12 will match any string of lowercase letters, underscores, and semicolons in quotes. Token 12 represents an Ada primary feature that requires an entry in the lex_spec file. The mk_lspec main driver routine first prints out the constant header information (Lex parameters) for the Lex specification. Then, for each recognized feature in the input grammar, an "expression   action" pair is printed. The action is to return a unique token number that is determined by the value of "Counter". After the entire grammar has been processed, the constant trailer information for the lex_spec file is printed out.

```
%%
\(                      { return(  1); }
\)                      { return(  2); }
\"                      { return(  3); }
\|                      { return(  4); }
\%                      { return(  5); }
\/                      { return(  6); }
\*                      { return(  7); }
\\                      { return(  8); }
\.                      { return(  9); }
\-                      { return( 10); }
\=                      { return( 11); }
\"[ ]*[a-z\_\:]+[ ]*\" { return( 12); }
[A-Za-z\_\:]+           { return( 13); }
[0-9]+                  { return( 14); }
[ ]+                    { return(998); }
[\n]                    { return(999); }
```

```
%%
#include <string.h>
main()
{
    int Token, Counter, Length;    /* DECLARE VARIABLES */
    char *Output;                  /* DECLARE OUTPUT STRING */
    Counter = 1;                   /* INTIALIZE TOKEN COUNTER */

    printf ("%s", "%a 6000  \n");  /* PRINT LEX PARAMETERS */
    printf ("%s", "%e 7000  \n");
    printf ("%s", "%n 4000  \n");
    printf ("%s", "%p 24000 \n");
    printf ("%s", "%o 5000  \n");
    printf ("%s", "%%       \n");

    while ((Token = yylex()) != 0) {   /* WHILE NOT END-OF-FILE... */
        if (Token == 12) {

            /* STRIP OF QUOTE MARKS AND PRINT EXPRESSION AND ACTION */

            Length = strlen (yytext);
            yytext[--Length] = '\0';
            Output = &yytext[1];
            printf ("%-40s{ return(%3u); }\n", Output, Counter);
            Counter++; }
    }

    /* PRINT REMAINING DEFAULT ENTRIES TO THE LEX_SPEC FILE */

    printf ("START_COMPILATION:                   { return(995); }\n");
    printf (":END_COMPILATION                     { return(996); }\n");
    printf ("[a-z\\:\\_]+                          { return(997); }\n");
    printf ("[ ]+                                 { return(998); }\n");
    printf ("[\\n]                                 { return(999); }\n");
    printf ("%s", "%%  \n");
}
```

Appendix D. *Source Code*

D.1 Shell Script

```
#-----------------------------------------------------------------------------
#--                              FILE HEADER                                --
#--                                                                         --
#--  DATE:  31 Aug 90                                                       --
#--  VERSION:  1.0                                                          --
#--  TITLE:  ALIANT Prototype Shell Script                                  --
#--  FILENAME:  runa                                                        --
#--  COORDINATOR:  Capt James S. Marr                                       --
#--  PROJECT:  GCS-90D Thesis                                               --
#--  OPERATING SYSTEM:  4.3 BSD UNIX                                        --
#--  LANGUAGE:  UNIX Shell Script                                           --
#--  FILE PROCESSING:  This file can be executed by entering 'csh runa'.    --
#--    To eliminate the requirement to enter 'csh', the file can be made    --
#--    independently executable by running the command 'chmod 755 runa'.   --
#--    After executing the 'chmod' command, the script can be executed by   --
#--    simply entering 'runa*'.                                             --
#--  CONTENTS:  This file contains the UNIX shell script that is used       --
#--    to execute the ALIANT prototype.                                     --
#--  FUNCTION:  This script provides the interface between the Gen test case --
#--    generator and the ALIANT Ada code.  Error checking is conducted on   --
#--    input parameters and informative error messages are produced when    --
#--    necessary.  There are two basic ways in which the ALIANT prototype is --
#--    executed.  The following algorithms illustrate the methods where     --
#--    parameter1 is the input grammar filename (minus '.gen'), parameter2  --
#--    is the requested number of combinations to generate, and parameter3  --
#--    is the batch input filename:                                         --
#--                                                                         --
#--        Method 1:                                                        --
#--            put <parameter1>.gen file into g_temp file                   --
#--            append "* <parameter2> compilation" to g_temp file           --
#--            execute gen.exe* with input from g_temp and                  --
#--              output directed to gen_out                                 --
#--            if gen.exe* terminated normally                              --
#--                overwrite contents of g_temp with <parameter2>           --
#--                execute aliant_driver.exe*                               --
#--                                                                         --
#--        Method 2:                                                        --
#--            put <parameter1>.gen file into g_temp file                   --
#--            append "* <parameter2> compilation" to g_temp file           --
#--            execute gen.exe* with input from g_temp and                  --
#--              output directed to gen_out                                 --
#--            if gen.exe* terminated normally                              --
#--                overwrite contents of g_temp with <parameter2>           --
#--                execute aliant_driver.exe* with input from               --
#--                  <parameter3> and output directed to alnt_out           --
#--                                                                         --
#--   If two valid input parameters are provided, the first method is used. --
#--   If three valid input parameters are provided, the second method is    --
#--   use.                                                                  --
#--                                                                         --
#----------------------------------------------------------------------------

## BEGIN SCRIPT ##

    ## OUTPUT DATE AND TIME TO ALNT_OUT FILE ##

    unset noclobber
    date > alnt_out
    set noclobber

    ## IF THERE ARE NO ARGUMENTS OR ONLY ONE, DISPLAY ERROR MESSAGE ##

    if ($#argv == 0 || $#argv == 1) then
        clear
        echo \*
        echo \* Missing Gen filename and/or combinations argument\(s\), try again.
        echo \*
        echo \*---- Format: runa\* fn1 num \[fn2\]
        echo \*---- Where : fn1.gen is the Gen input file,
        echo \*------------ num is the desired number of combinations, and
        echo \*------------ fn2 is an optional ALIANT batch input file.
        echo \*

    ## IF THERE ARE TWO ARGUMENTS, CONTINUE PROCESSING SCRIPT ##

    else if ($#argv == 2) then
        clear

        ## IF THE TWO ARGUMENTS ARE VALID, EXECUTE ALIANT PROTOTYPE ##

        if (-e $argv[1].gen && $argv[2] > 0) then
            unset noclobber
            echo -------------------------------
            echo -- Gen execution in progress --
            echo -------------------------------
            cat $argv[1].gen > g_temp
            echo \* $argv[2] compilation >> g_temp
            gen.exe*<g_temp>gen_out && echo $argv[2]>g_temp && aliant_driver.exe*
            set noclobber

        ## IF BOTH ARGUMENTS ARE NOT VALID, DISPLAY AN ERROR MESSAGE ##

        else
            echo \*
            echo \* Gen filename provided does not exist and/or number
            echo \* of combinations provided not greater than 0, try again.
            echo \*
        endif

    ## IF THERE ARE THREE ARGUMENTS, CONTINUE PROCESSING SCRIPT ##

    else if ($#argv == 3) then

        ## IF ALL THREE ARGUMENTS ARE VALID, EXECUTE ALIANT PROTOTYPE ##

        if (-e $argv[1].gen == 1 && $argv[2] > 0 && -e $argv[3] == 1) then
            unset noclobber
            cat $argv[1].gen > g_temp
            echo \* $argv[2] compilation >> g_temp
            gen.exe*<g_temp>gen_out && echo $argv[2]>g_temp &&
                aliant_driver.exe*<$argv[3]>>alnt_out
            set noclobber

        ## IF ALL THREE ARGUMENTS ARE NOT VALID, DISPLAY AN ERROR MESSAGE ##

        else
            clear
            echo \*
            echo \* Gen filename provided does not exist and/or number
            echo \* of combinations provided not greater than 0 and/or
            echo \* ALIANT filename provided does not exist, try again.
            echo \*
        endif

    ## IF THERE ARE MORE THAN THREE ARGUMENTS, DISPLAY AN ERROR MESSAGE ##

    else
        clear
        echo \*
        echo \* Too many arguments provided, try again.
        echo \*
        echo \*---- Format: runa\* fn1 num \[fn2\]
        echo \*---- Where : fn1.gen is the Gen input file,
        echo \*------------ num is the desired number of combinations, and
        echo \*------------ fn2 is an optional ALIANT batch input file.
        echo \*
    endif

    ## OUTPUT DATE AND TIME TO THE ALNT_OUT FILE ##

    date >> alnt_out

## END OF SCRIPT ##
```

D.2 Ada Code

```
-----------------------------------------------------------------------------
--                            FILE HEADER                                  --
--                                                                         --
-- DATE:  31 Aug 90                                                        --
-- VERSION:  1.0                                                           --
-- TITLE:  ALIANT Prototype                                                --
-- FILENAME:  aliant.a                                                     --
-- COORDINATOR:  Capt James S. Marr                                        --
-- PROJECT:  GCS-90D Thesis                                                --
-- OPERATING SYSTEM:  4.3 BSD UNIX                                         --
-- LANGUAGE:  Elxsi Verdix Ada (Version 5.5)                               --
-- FILE PROCESSING:  This file is compiled using the Verdix command string --
--   'ada aliant.a'.  The Text_IO and Math library packages are required   --
--   for compilation and linking.  The object code is linked using the     --
--   Verdix command string 'a.ld aliant_driver -o aliant_driver.exe'.      --
--   The filename 'yylex' must also be available for linking.  This file   --
--   contains, among other things, the C procedures 'yylex', 'opengen',    --
--   and 'closegen'.  These procedures are linked to the Ada code using    --
--   pragma interface.  The Verdix 'ada.lib' file must contain a link      --
--   entry for the 'yylex' file.  Further details are included in the      --
--   documentation for the Lex_Pkg source code.                            --
-- CONTENTS:                                                               --
--   Lex_Pkg - Ada package that provides interface to C procedures.        --
--   Parameter_Pkg - Ada package that contains parameters used throughout  --
--     the ALIANT prototype.                                               --
--   Features_Pkg - Ada package that contains the Ada features table.      --
--   Matrix_Pkg - Ada package that contains the combination storage        --
--     matrix and associated access procedures.                            --
--   ALIANT_Driver - Ada procedure that controls the ALIANT execution.     --
-- FUNCTION:  This file contains all the Ada code supporting the ALIANT    --
--   prototype.  This code works in conjunction with a test case generator --
--   (written in C) via intermediate ASCII data files.  The control of     --
--   the interface is handled by a UNIX shell script.  The shell script    --
--   executes the test case generator, storing the results in a data file. --
--   Providing the test case generator terminated normally, the ALIANT     --
--   driver is executed to analyze the output stored in the intermediate   --
--   data file.                                                            --
--                                                                         --
-----------------------------------------------------------------------------

-----------------------------------------------------------------------------
--                           PACKAGE HEADER                                --
--                                                                         --
-- DATE:  31 Aug 90                                                        --
-- VERSION:  1.0                                                           --
-- NAME:  Lex_Pkg                                                          --
-- PACKAGE TYPE:  Specification only.                                      --
-- CONTENTS:  Specification of function Yylex and procedures Opengen and   --
--    Closegen.                                                         --
--  DESCRIPTION:  This package specifies the interface to the three    --
--    subunits mentioned above using 'pragma interface'.  These subunits --
--    are written in the C language and linked to the ALIANT prototype.  --
--    Assuming the object code for Yylex, Opengen, and Closegen is located --
--    in a file called 'yylex'; the following entry in the Verdix library --
--    (ada.lib) will allow these routines to be linked with the Ada code: --
--    'WITH1:LINK:yylex;'.                                               --
--  ENCAPSULATED OBJECTS:  None.                                         --
--  OBJECT OPERATORS:  None.                                             --
--  FILES READ:  None.                                                   --
--  FILES WRITTEN:  None.                                                --
--  HARDWARE INPUT:  None.                                               --
--  HARDWARE OUTPUT:  None.                                              --
--  REQUIRED LIBRARY UNITS:  None.                                       --
--  CALLING MODULES:                                                     --
--    Yylex called by:  ALIANT_Driver                                    --
--    Opengen called by:  ALIANT_Driver                                  --
--    Closegen called by: ALIANT_Driver.ALIANT_Wrapup                    --
--                                                                       --
--  AUTHOR:  Capt James S. Marr                                          --
--  HISTORY:  None.                                                      --
--                                                                       --
---------------------------------------------------------------------------

package Lex_Pkg is

    function Yylex return integer;
    procedure Opengen;
    procedure Closegen;

private

    pragma interface (C, Yylex);
    pragma interface (C, Opengen);
    pragma interface (C, Closegen);

end Lex_Pkg;

---------------------------------------------------------------------------
--                          PACKAGE HEADER                               --
--                                                                       --
--  DATE:  31 Aug 90                                                     --
--  VERSION:  1.0                                                        --
--  NAME:  Parameter_Pkg                                                 --
--  PACKAGE TYPE:  Specification and Body.                               --
--  CONTENTS:  This package declares several constant and variable       --
--    parameters and types used throughout the ALIANT prototype,         --
--    a Screen_Delay procedure, and two ALIANT-unique exceptions.        --
--  DESCRIPTION:  The specification contains the constants requiring     --
--    visibility throughout the ALIANT prototype.  Two of the parameters --
--    are accessed via function calls to Get_Max_Features and          --
--    Get_Max_Combinations. These two encapsulated objects are initialized --
--    when the Parameter_Pkg body is elaborated.                        --
--    The Max_Features is initialized by reading the lex_spec file to   --
--    determine how many features there are. The Max_Combinations is   --
--    initialized by reading the g_temp file to determine how many      --
--    combinations were requested by the user. A calculation is then made --
--    to determine the approximate storage space that will be required to --
--    hold the expected number of unique combinations generated.        --
--  ENCAPSULATED OBJECTS: Max_Features and Max_Combinations.            --
--  OBJECT OPERATORS: Get_Max_Features and Get_Max_Combinations.        --
--  FILES READ: Lex_Spec_File and Gen_Combination_File.                 --
--  FILES WRITTEN: None.                                                --
--  HARDWARE INPUT: File input.                                         --
--  HARDWARE OUTPUT: CRT.                                               --
--  REQUIRED LIBRARY UNITS: Body requires Text_IO and Math.             --
--  MODULES CALLED (by executable package body):                        --
--    Text_IO.new_page                                                  --
--    Text_IO.new_line                                                  --
--    Text_IO.put_line                                                  --
--    Text_IO.open                                                      --
--    Math.sqrt                                                         --
--    Text_IO.get_line                                                  --
--    Text_IO.skip_line                                                 --
--    Text_IO.close                                                     --
--    Combination_IO.get (instantiation of Text_IO.integer_io)          --
--                                                                      --
--  AUTHOR: Capt James S. Marr                                          --
--  HISTORY: None.                                                      --
--                                                                      --
--------------------------------------------------------------------------

package Parameter_Pkg is

    Feature_Length            : constant := 40;
    Lex_Spec_Filename         : constant string (1..8) := "lex_spec";
    Database_Filename         : constant string (1..7) := "afis_db";
    Gen_Combination_Filename : constant string (1..6) := "g_temp";

    subtype Feature_String is string (1..Feature_Length);
    subtype Parameter_Type is integer range 1..3500;

    procedure Screen_Delay;
    function Get_Max_Features return Parameter_Type;
    function Get_Max_Combinations return Parameter_Type;

    Fatal_Exception   : exception;
    Partial_Exception : exception;

end Parameter_Pkg;
```

```
with text_io;
with math;
package body Parameter_Pkg is

    Max_Features          : Parameter_Type;
    Max_Combinations      : Parameter_Type;
    Input_Combinations    : natural;
    Feature_Count         : natural;
    String_Length         : natural;
    Lex_Spec_File         : text_io.file_type;
    Gen_Combination_File  : text_io.file_type;
    Input_String          : Parameter_Pkg.Feature_String := (others => ' ');

-----------------------------------------------------------------------------
--                             MODULE HEADER                               --
--                                                                         --
--  DATE:  31 Aug 90                                                       --
--  VERSION:  1.0                                                          --
--  NAME:  Screen_Delay                                                    --
--  DESCRIPTION:  This procedure is used in several places in the ALIANT   --
--    prototype to provide a time delay for displaying user input error    --
--    messages on the terminal screen.                                     --
--  ALGORITHM:  Execute the Ada delay statement.                           --
--  PASSED VARIABLES:  None.                                               --
--  RETURNS:  None                                                         --
--  GLOBAL VARIABLES USED:  None.                                          --
--  GLOBAL VARIABLES CHANGED:  None.                                       --
--  FILES READ:  None.                                                     --
--  FILES WRITTEN:  None.                                                  --
--  HARDWARE INPUT:  None.                                                 --
--  HARDWARE OUTPUT:  None.                                                --
--  MODULES CALLED:  None.                                                 --
--  CALLING MODULES:                                                       --
--    Matrix_Pkg.Display_Matrix                                            --
--    Matrix_Pkg.Load_Database                                             --
--                                                                         --
--  AUTHOR:  Capt James S. Marr                                            --
--  HISTORY:  None.                                                        --
--                                                                         --
--  ORDER-OF ANALYSIS: O(1) since it just executes a single statement.     --
--                                                                         --
-----------------------------------------------------------------------------

    procedure Screen_Delay is

    begin

        delay (duration (1.5));

    end Screen_Delay;
```

```
------------------------------------------------------------------------------
--                            MODULE HEADER                                 --
--                                                                          --
-- DATE:  31 Aug 90                                                         --
-- VERSION:  1.0                                                            --
-- NAME:  Get_Max_Features                                                  --
-- DESCRIPTION:  This function is used to get the current value of          --
--   Parameter_Pkg.Max_Features.                                            --
-- ALGORITHM:  Return the encapsulated package variable Max_Features.       --
-- PASSED VARIABLES:  None.                                                 --
-- RETURNS:  Parameter_Pkg.Max_Features                                     --
-- GLOBAL VARIABLES USED:  Parameter_Pkg.Max_Features                       --
-- GLOBAL VARIABLES CHANGED:  None.                                         --
-- FILES READ:  None.                                                       --
-- FILES WRITTEN:  None.                                                    --
-- HARDWARE INPUT:  None.                                                   --
-- HARDWARE OUTPUT:  None.                                                  --
-- MODULES CALLED:  None.                                                   --
-- CALLING MODULES:                                                         --
--   Declarative parts in bodies of:                                        --
--     Features_Pkg                                                         --
--     Matrix_Pkg                                                           --
--     ALIANT_Driver                                                        --
--                                                                          --
-- AUTHOR:  Capt James S. Marr                                              --
-- HISTORY:  None.                                                          --
--                                                                          --
-- ORDER-OF ANALYSIS:  O(1) since just a single statement is executed.      --
--                                                                          --
------------------------------------------------------------------------------

    function Get_Max_features return Parameter_Type is

    begin

        return (Max_Features);

    end Get_Max_Features;

------------------------------------------------------------------------------
--                            MODULE HEADER                                 --
--                                                                          --
-- DATE:  31 Aug 90                                                         --
-- VERSION:  1.0                                                            --
-- NAME:  Get_Max_Combinations                                              --
-- DESCRIPTION:  This function is used to get the current value of          --
--   Parameter_Pkg.Max_Combinations.                                        --
-- ALGORITHM:  Return the encapsulated package variable Max_Combinations.   --
-- PASSED VARIABLES:  None.                                                 --
-- RETURNS:  Parameter_Pkg.Max_Combinations                                 --
-- GLOBAL VARIABLES USED:  Parameter_Pkg.Max_Combinations                   --
-- GLOBAL VARIABLES CHANGED:  None.                                        --
-- FILES READ:  None.                                                     --
-- FILES WRITTEN:  None.                                                  --
-- HARDWARE INPUT:  None.                                                 --
-- HARDWARE OUTPUT:  None.                                                --
-- MODULES CALLED:  None.                                                 --
-- CALLING MODULES:  Declarative part in body of Matrix_Pkg.              --
--                                                                        --
-- AUTHOR:  Capt James S. Marr                                            --
-- HISTORY:  None.                                                        --
--                                                                        --
-- ORDER-OF ANALYSIS:  O(1) since just a single statement is executed.    --
--                                                                        --
----------------------------------------------------------------------------

    function Get_Max_Combinations return Parameter_Type is

    begin

        return (Max_Combinations);

    end Get_Max_Combinations;


    package Combination_IO is new text_io.integer_io (natural);

begin

    -- DISPLAY OPENING ALIANT MESSAGE --

    text_io.new_page;
    text_io.new_line;
    text_io.put_line (" ***************************************");
    text_io.put_line (" ** ALIANT initialization in progress **");
    text_io.put_line (" ***************************************");
    text_io.new_line;

    -- BEGIN BLOCK TO DETERMINE VALUE FOR MAX_FEATURES --

    begin

        -- OPEN LEX_SPEC_FILE AND SKIP LEX PARAMETER LINES --

        text_io.open (Lex_Spec_File, text_io.in_file,
                      Parameter_Pkg.Lex_Spec_Filename);
        while (Input_String (1..2) /= "%%") loop
            text_io.get_line (Lex_Spec_File, Input_String, String_Length);
        end loop;

        -- COUNT THE NUMBER OF FEATURES IN THE LEX_SPEC_FILE --
    Feature_Count := 0;
    while (Input_String (1..18) /= "START_COMPILATION:") loop
        text_io.get_line (Lex_Spec_File, Input_String, String_Length);
        text_io.skip_line (Lex_Spec_File);
        Feature_Count := Feature_Count + 1;
    end loop;
    text_io.close (Lex_Spec_File);
    Max_Features := Feature_Count - 1;

exception
    when text_io.name_error =>
        text_io.new_line;
        text_io.put_line (" <Parameter_Pkg body>");
        text_io.put_line (" *** NAME EXCEPTION ERROR RAISED WHILE  ***");
        text_io.put_line (" *** TRYING TO OPEN LEX SPECIFICATION   ***");
        text_io.put_line (" *** FILE.  CHECK FILENAME IN PARAMETER ***");
        text_io.put_line (" *** PACKAGE AND CURRENT DIRECTORY.     ***");
        text_io.new_line;
        raise Parameter_Pkg.Fatal_Exception;
    when text_io.end_error =>
        text_io.new_line;
        text_io.put_line (" <Parameter_Pkg body>");
        text_io.put_line (" *** PREMATURE END-OF-FILE REACHED WHILE ***");
        text_io.put_line (" *** READING LEX SPECIFICATION FILE.     ***");
        text_io.put_line (" *** CHECK FORMAT OF LEX SPECIFICATION.  ***");
        text_io.new_line;
        text_io.close (Lex_Spec_File);
        raise Parameter_Pkg.Fatal_Exception;
    when constraint_error =>
        text_io.new_line;
        text_io.put_line (" <Parameter_Pkg body>");
        text_io.put_line (" *** NUMBER OF FEATURES IS OUT OF RANGE.  ***");
        text_io.put_line (" *** CHECK LEX SPECIFICATION FILE AND     ***");
        text_io.put_line (" *** PARAMETER TYPE IN PARAMETER PACKAGE. ***"),
        text_io.new_line;
        raise Parameter_Pkg.Fatal_Exception;
    when others =>
        text_io.new_line;
        text_io.put_line (" <Parameter_Pkg body.1>");
        text_io.put_line (" *** UNKNOWN EXCEPTION RAISED   ***");
        text_io.put_line (" *** WHILE INITIALIZING ALIANT. ***");
        text_io.new_line;
        raise;
end;

-- BEGIN BLOCK TO DETERMINE VALUE FOR MAX_COMBINATIONS --

begin

    -- GET INPUT_COMBINATIONS FROM GEN_COMBINATION_FILE --
    text_io.open (Gen_Combination_File, text_io.in_file,
                  Parameter_Pkg.Gen_Combination_Filename);
    Combination_IO.get (Gen_Combination_File, Input_Combinations);
    text_io.close (Gen_Combination_File);

    -- CALCULATE AN ESTIMATED VALUE FOR MAX_COMBINATIONS BASED --
    -- ON THE VALUE OF INPUT_COMBINATIONS.  THIS CALCULATION   --
    -- REDUCES THE AMOUNT OF UNUSED SPACE IN THE COMBINATION   --
    -- MATRIX FOR LARGE VALUES OF INPUT_COMBINATIONS.          --

    Max_Combinations := integer (
      math.sqrt (1200.0 * float (Input_Combinations)));

exception
    when text_io.name_error =>
        text_io.new_line;
        text_io.put_line (" <Parameter_Pkg body>");
        text_io.put_line (" *** NAME EXCEPTION ERROR RAISED WHILE  ***");
        text_io.put_line (" *** TRYING TO OPEN THE GEN COMBINATION ***");
        text_io.put_line (" *** FILE.  CHECK FILENAME IN PARAMETER ***");
        text_io.put_line (" *** PACKAGE AND CURRENT DIRECTORY.     ***");
        text_io.new_line;
        raise Parameter_Pkg.Fatal_Exception;
    when text_io.end_error =>
        text_io.new_line;
        text_io.put_line (" <Parameter_Pkg body>");
        text_io.put_line (" *** PREMATURE END-OF-FILE REACHED WHILE ***");
        text_io.put_line (" *** READING GEN COMBINATION FILE. CHECK ***");
        text_io.put_line (" *** FORMAT OF GEN COMBINATION FILE.     ***");
        text_io.new_line;
        text_io.close (Gen_Combination_File);
        raise Parameter_Pkg.Fatal_Exception;
    when text_io.data_error =>
        text_io.new_line;
        text_io.put_line (" <Parameter_Pkg body.1>");
        text_io.put_line (" *** NUMBER OF COMBINATIONS IS OUT OF      ***");
        text_io.put_line (" *** RANGE. CHECK GEN COMBINATION FILE AND ***");
        text_io.put_line (" *** PARAMETER TYPE IN PARAMETER PACKAGE.  ***");
        text_io.new_line;
        text_io.close (Gen_Combination_File);
        raise Parameter_Pkg.Fatal_Exception;
    when constraint_error =>
        text_io.new_line;
        text_io.put_line (" <Parameter_Pkg body.2>");
        text_io.put_line (" *** NUMBER OF COMBINATIONS IS OUT OF      ***");
        text_io.put_line (" *** RANGE. CHECK GEN COMBINATION FILE AND ***");
        text_io.put_line (" *** PARAMETER TYPE IN PARAMETER PACKAGE.  ***");
        text_io.new_line;
        raise Parameter_Pkg.Fatal_Exception;
    when others =>
        text_io.new_line;
```

```
          text_io.put_line (" <Parameter_Pkg body.2>");
          text_io.put_line (" *** UNKNOWN EXCEPTION RAISED   ***");
          text_io.put_line (" *** WHILE INITIALIZING ALIANT. ***");
          text_io.new_line;
          raise;
    end;

end Parameter_Pkg;

--------------------------------------------------------------------------------
--                            PACKAGE HEADER                                  --
--                                                                            --
-- DATE:  31 Aug 90                                                           --
-- VERSION:  1.0                                                              --
-- NAME:  Features_Pkg                                                        --
-- PACKAGE TYPE:  Specification and Body.                                     --
-- CONTENTS:  This package contains a procedure Load_Features_Table and a    --
--   function Get_Feature.                                                    --
-- DESCRIPTION:  The procedure is used to load the Ada features from the     --
--   lex_spec file into an array of text strings.  The function is used      --
--   within the ALIANT prototype to access the Ada features.                 --
-- ENCAPSULATED OBJECTS:  Features_Table                                      --
-- OBJECT OPERATORS:  Get_Feature                                             --
-- FILES READ:  None.                                                         --
-- FILES WRITTEN:  None.                                                      --
-- HARDWARE INPUT:  None.                                                     --
-- HARDWARE OUTPUT:  None.                                                    --
-- REQUIRED LIBRARY UNITS:  Specification requires Parameter_Pkg and         --
--   Body requires Text_IO and Parameter_Pkg.  The pragma elaborate is       --
--   used with the Parameter_Pkg.                                             --
--                                                                            --
-- AUTHOR.  Capt James S. Marr                                                --
-- HISTORY:  None.                                                            --
--                                                                            --
--------------------------------------------------------------------------------

with Parameter_Pkg;
package Features_Pkg is

    procedure Load_Features_Table;
    function Get_Feature (Feature_Number : in natural)
      return Parameter_Pkg.Feature_String;

end Features_Pkg;

with text_io;
with Parameter_Pkg;
pragma elaborate (Parameter_Pkg);
package body Features_Pkg is

    Max_Features   : constant Parameter_Pkg.Parameter_Type
    := Parameter_Pkg.Get_Max_Features;
  Features_Table : array (1..Max_Features) of
    Parameter_Pkg.Feature_String;

-----------------------------------------------------------------------------
--                              MODULE HEADER                              --
--                                                                         --
-- DATE:  31 Aug 90                                                        --
-- VERSION:  1.0                                                           --
-- NAME:  Load_Features_Table                                              --
-- DESCRIPTION:  This procedure loads the Ada features from the lex_spec   --
--   file into the Features_Table.                                         --
-- ALGORITHM:                                                              --
--   open lex_spec file                                                    --
--   skip Lex parameter lines                                              --
--   while not end-of-file lex_spec file                                   --
--       read next line of lex_spec file                                   --
--       strip off Ada feature and store in Features_Table                 --
--   close lex_spec file                                                   --
-- PASSED VARIABLES:  None.                                                --
-- RETURNS:  None.                                                         --
-- GLOBAL VARIABLES USED:  Parameter_Pkg.Lex_Spec_Filename                 --
-- GLOBAL VARIABLES CHANGED:  Features_Pkg.Features_Table                  --
-- FILES READ:  Lex_Spec_File                                              --
-- FILES WRITTEN:  None.                                                   --
-- HARDWARE INPUT:  File input.                                            --
-- HARDWARE OUTPUT: None.                                                  --
-- MODULES CALLED:                                                         --
--   Text_IO.open                                                          --
--   Text_IO.get_line                                                      --
--   Text_IO.close                                                         --
-- CALLING MODULES: ALIANT_Driver.                                         --
--                                                                         --
-- AUTHOR:  Capt James S. Marr                                             --
-- HISTORY:  None.                                                         --
--                                                                         --
-- ORDER-OF ANALYSIS:  O(n) since the procedure is dominated by a loop     --
--   that executes once for each feature.  Therefore, n is dependent on    --
--   the value of Max_Features.                                            --
--                                                                         --
-----------------------------------------------------------------------------

  procedure Load_Features_Table is

      String_Length : natural;
      Lex_Spec_File : text_io.file_type;
      Blanks        : constant Parameter_Pkg.Feature_String
                      := (others => ' ');
      Input_String  : Parameter_Pkg.Feature_String := Blanks;

  begin
```

```
        -- OPEN LEX_SPEC_FILE AND SKIP LEX PARAMETER LINES --

        text_io.open (Lex_Spec_File, text_io.in_file,
                      Parameter_Pkg.Lex_Spec_Filename);
        while (Input_String (1..2) /= "%%") loop
            text_io.get_line (Lex_Spec_File, Input_String, String_Length);
        end loop;

        -- LOAD EACH FEATURE INTO THE FEATURES_TABLE --

        for I in 1..Max_Features loop
            Input_String := Blanks;
            text_io.get_line (Lex_Spec_File, Input_String, String_Length);
            text_io.skip_line (Lex_Spec_File);
            Features_Table (I) := Input_String;
        end loop;

        text_io.close (Lex_Spec_File);

    exception
        when text_io.name_error =>
            text_io.new_line;
            text_io.put_line (" <Features_Pkg.Load_Features_Table>");
            text_io.put_line (" *** NAME EXCEPTION ERROR RAISED WHILE  ***");
            text_io.put_line (" *** TRYING TO OPEN LEX SPECIFICATION   ***");
            text_io.put_line (" *** FILE.  CHECK FILENAME IN PARAMETER ***");
            text_io.put_line (" *** PACKAGE AND CURRENT DIRECTORY.     ***");
            text_io.new_line;
            raise Parameter_Pkg.Fatal_Exception;
        when text_io.end_error =>
            text_io.new_line;
            text_io.put_line (" <Features_Pkg.Load_Features_Table>");
            text_io.put_line (" *** PREMATURE END-OF-FILE REACHED WHILE ***");
            text_io.put_line (" *** READING LEX SPECIFICATION FILE.     ***");
            text_io.put_line (" *** CHECK FORMAT OF LEX SPECIFICATION.  ***");
            text_io.new_line;
            text_io.close (Lex_Spec_File);
            raise Parameter_Pkg.Fatal_Exception;

    end Load_Features_Table;

------------------------------------------------------------------------------
--                               MODULE HEADER                              --
--                                                                          --
-- DATE:  31 Aug 90                                                         --
-- VERSION:  1.0                                                            --
-- NAME:  Get_Feature                                                       --
-- DESCRIPTION:  This function is used to get a feature string for a        --
--   specified feature number.                                              --
-- ALGORITHM:  Return requested feature string.                             --
-- PASSED VARIABLES:  Feature_Number.                                        --
-- RETURNS:  Feature_String.                                                 --
-- GLOBAL VARIABLES USED:  Features_Pkg.Features_Table.                      --
-- GLOBAL VARIABLES CHANGED:  None.                                          --
-- FILES READ:  None.                                                        --
-- FILES WRITTEN:  None.                                                     --
-- HARDWARE INPUT:  None.                                                    --
-- HARDWARE OUTPUT:  None.                                                   --
-- MODULES CALLED:  None.                                                    --
-- CALLING MODULES:  Matrix_Pkg.Display_Matrix                               --
--                                                                           --
-- AUTHOR:  Capt James S. Marr                                               --
-- HISTORY:  None.                                                           --
--                                                                           --
-- ORDER-OF ANALYSIS:  O(1) since just a single statement is executed.       --
--                                                                           --
-------------------------------------------------------------------------------

    function Get_Feature (Feature_Number : in natural)
      return Parameter_Pkg.Feature_String is

    begin

        return Features_Table (Feature_Number);

    end Get_Feature;

end Features_Pkg;

-------------------------------------------------------------------------------
--                              PACKAGE HEADER                               --
--                                                                           --
-- DATE:  31 Aug 90                                                          --
-- VERSION:  1.0                                                             --
-- NAME:  Matrix_Pkg                                                         --
-- PACKAGE TYPE:  Specification and Body.                                    --
-- CONTENTS:  This package contains six procedures used to access the        --
--   Combination_Matrix.                                                     --
-- DESCRIPTION:  When the body is elaborated, the declarative part creates --
--   the Combination_Matrix that is used to store the feature counts for     --
--   each combination identified in the Gen input file.  This package        --
--   includes all the necessary procedures to initialize, update, and        --
--   retrieve the contents of the Combination_Matrix.                        --
-- ENCAPSULATED OBJECTS:  Combination_Matrix                                 --
-- OBJECT OPERATORS:  Initialize_Matrix, Start_Combination, Count_Feature, --
--   End_Combination, Display_Matrix, and Load_Database.                     --
-- FILES READ:  None.                                                        --
-- FILES WRITTEN:  None.                                                     --
-- HARDWARE INPUT:  None.                                                    --
-- HARDWARE OUTPUT:  None.                                                   --
-- REQUIRED LIBRARY UNITS:  Body requires Text_IO, Features_Pkg, and         --
--    Parameter_Pkg.  Pragma elaborate is used with the Parameter_Pkg.      --
--                                                                           --
--  AUTHOR:  Capt James S. Marr                                              --
--  HISTORY:  None.                                                          --
--                                                                           --
-------------------------------------------------------------------------------

package Matrix_Pkg is

    procedure Initialize_Matrix;
    procedure Start_Combination;
    procedure Count_Feature (Feature_Number : in natural);
    procedure End_Combination;
    procedure Display_Matrix;
    procedure Load_Database;

end Matrix_Pkg;

with text_io;
with Features_Pkg;
with Parameter_Pkg;
pragma elaborate (Parameter_Pkg);
package body Matrix_Pkg is

    Max_Features     : constant Parameter_Pkg.Parameter_Type
      := Parameter_Pkg.Get_Max_Features;
    Max_Combinations : constant Parameter_Pkg.Parameter_Type
      := Parameter_Pkg.Get_Max_Combinations;
    subtype Comb_Number_Type is integer
      range 0..Max_Combinations + 1;
    Combination_Matrix : array (1..Max_Combinations,
                                -1..Max_Features) of natural;
    Current_Comb          : Comb_Number_Type := 0;
    Next_Comb             : Comb_Number_Type := 1;
    Duplicate_Count       : integer := 0;
    Null_Count            : integer := 0;
    Number_Comb_Processed : integer := 0;

    package Natural_IO is new text_io.integer_io (natural);

-------------------------------------------------------------------------------
--                            MODULE HEADER                                  --
--                                                                           --
--  DATE:  31 Aug 90                                                         --
--  VERSION:  1.0                                                            --
--  NAME:  Initialize_Matrix                                                 --
--  DESCRIPTION:  This procedure performs the initialization of the          --
--    Combination_Matrix.                                                    --
--  ALGORITHM:  Using two nested loops, initialize each position of the      --
--    Combination_Matrix to zero.                                            --
--  PASSED VARIABLES:  None.                                                 --
-- RETURNS:  None.                                                       --
-- GLOBAL VARIABLES USED:  Matrix_Pkg.Max_Combinations and              --
--   Matrix_Pkg.Max_Features.                                            --
-- GLOBAL VARIABLES CHANGED:  Matrix_Pkg.Combination_Matrix              --
-- FILES READ:  None.                                                    --
-- FILES WRITTEN:  None.                                                 --
-- HARDWARE INPUT:  None.                                                --
-- HARDWARE OUTPUT:  None.                                               --
-- MODULES CALLED:  None.                                                --
-- CALLING MODULES:  ALIANT_Driver                                       --
--                                                                       --
-- AUTHOR:  Capt James S. Marr                                           --
-- HISTORY:  None.                                                       --
--                                                                       --
-- ORDER-OF ANALYSIS:  O(n**2) since the procedure contains two nested   --
--   loops.  The value of n**2 is actually the product of Max_Features and --
--   Max_Combinations.                                                   --
--                                                                       --
---------------------------------------------------------------------------

   procedure Initialize_Matrix is

   begin

       -- INITIALIZE ALL POSITIONS OF COMBINATION_MATRIX TO ZERO --

       for I in 1..Max_Combinations loop
           for J in -1..Max_Features loop
               Combination_Matrix (I,J) := 0;
           end loop;
       end loop;

   end Initialize_Matrix;

---------------------------------------------------------------------------
--                             MODULE HEADER                             --
--                                                                       --
-- DATE:  31 Aug 90                                                      --
-- VERSION:  1.0                                                         --
-- NAME:  Start_Combination                                              --
-- DESCRIPTION:  This procedure is executed whenever a new combination is --
--   identified.  Its purpose is to set appropriate counters and provide --
--   a runtime indication that processing is still in progress.          --
-- ALGORITHM:  Increment Current_Comb and Next_Comb and display a dot on  --
--   the user terminal for every 10 combinations identified.             --
-- PASSED VARIABLES:  None.                                              --
-- RETURNS:  None.                                                       --
-- GLOBAL VARIABLES USED:                                                --
--   Matrix_Pkg.Current_Comb                                             --
--   Matrix_Pkg.Next_Comb                                                --
--   Matrix_Pkg.Number_Comb_Processed                                    --
--  GLOBAL VARIABLES CHANGED:                                             --
--    Matrix_Pkg.Current_Comb                                             --
--    Matrix_Pkg.Next_Comb                                                --
--    Matrix_Pkg.Number_Comb_Processed                                    --
--  FILES READ:  None.                                                    --
--  FILES WRITTEN:  None.                                                 --
--  HARDWARE INPUT:  None.                                                --
--  HARDWARE OUTPUT: CRT.                                                 --
--  MODULES CALLED:                                                       --
--    Text_IO.put                                                         --
--    Text_IO.new_line                                                    --
--    Text_IO.put_line                                                    --
--  CALLING MODULES: ALIANT_Driver                                        --
--                                                                        --
--  AUTHOR:  Capt James S. Marr                                           --
--  HISTORY:  None.                                                       --
--                                                                        --
--  ORDER-OF ANALYSIS:  O(1) since only sequential statements are executed. --
--                                                                        --
----------------------------------------------------------------------------

  procedure Start_Combination is

  begin

    -- INCREMENT COMBINATION COUNTERS AND DISPLAY --
    -- A DOT FOR EVERY TEN COMBINATIONS PROCESSED --

    Current_Comb := Next_Comb;
    Next_Comb := Next_Comb + 1;
    if ((Number_Comb_Processed mod 10) = 0) then
        text_io.put (".");
    end if;
    Number_Comb_Processed := Number_Comb_Processed + 1;

  exception
    when constraint_error =>
        text_io.new_line;
        text_io.put_line (" <Matrix_Pkg.Start_Combination>");
        text_io.put_line (" *** TOO MANY GEN COMBINATIONS. ***");
        text_io.put_line (" *** PARTIAL RESULTS FOLLOW.    ***");
        text_io.new_line;
        Current_Comb := Current_Comb - 1;
        raise Parameter_Pkg.Partial_Exception;

   end Start_Combination;

----------------------------------------------------------------------------
--                              MODULE HEADER                             --
--                                                                        --
--  DATE:  31 Aug 90                                                      --
-- VERSION:  1.0                                                          --
-- NAME:  Count_Feature                                                   --
-- DESCRIPTION:  This procedure is used to increment a specified feature  --
--   count for the current combination being processed.                   --
-- ALGORITHM:  Increment feature count by subscripting Combination_Matrix --
--   with Current_Comb and Feature_Number.                                --
-- PASSED VARIABLES:  Feature_Number                                      --
-- RETURNS:  None.                                                        --
-- GLOBAL VARIABLES USED:  Matrix_Pkg.Current_Comb                        --
-- GLOBAL VARIABLES CHANGED:  Matrix_Pkg.Combination_Matrix               --
-- FILES READ:  None.                                                     --
-- FILES WRITTEN:  None.                                                  --
-- HARDWARE INPUT:  None.                                                 --
-- HARDWARE OUTPUT:  CRT (exception messages)                             --
-- MODULES CALLED:  Text_IO.new_line, Text_IO.put_line (exception)        --
-- CALLING MODULES:  ALIANT_Driver                                        --
--                                                                        --
-- AUTHOR:  Capt James S. Marr                                            --
-- HISTORY:  None.                                                        --
--                                                                        --
-- ORDER-OF ANALYSIS:  O(1) since only sequential statements executed.    --
--                                                                        --
------------------------------------------------------------------------------

   procedure Count_Feature (Feature_Number : in natural) is

   begin

       -- INCREMENT APPROPRIATE FEATURE COUNT BY ONE --

       Combination_Matrix(Current_Comb,Feature_Number) :=
         Combination_Matrix(Current_Comb,Feature_Number) + 1;

   exception
       when constraint_error =>
           text_io.new_line;
           if (Current_Comb = 0) then
               text_io.put_line (" <Matrix_Pkg.Count_Feature>");
               text_io.put_line (" *** INCORRECT FORMAT IN GEN INPUT ***");
               text_io.put_line (" *** FILE.  CHECK THE GEN GRAMMAR. ***");
               text_io.new_line;
               raise Parameter_Pkg.Fatal_Exception;
           else
               text_io.put_line (" <Matrix_Pkg.Count_Feature>");
               text_io.put_line (" *** FEATURE NUMBER OUT OF RANGE IN  ***");
               text_io.put_line (" *** COUNT_FEATURE.  CHECK LEX SPEC  ***");
               text_io.put_line (" *** AND PARAMETER PACKAGE.          ***");
               text_io.new_line;
               raise Parameter_Pkg.Fatal_Exception;
           end if;
   end Count_Feature;

--------------------------------------------------------------------------------
--                              MODULE HEADER                                 --
--                                                                            --
-- DATE:  31 Aug 90                                                           --
-- VERSION:  1.0                                                              --
-- NAME:  End_Combination                                                     --
-- DESCRIPTION:  This procedure is executed whenever the end of a             --
--   combination is detected.  The ended combination is checked to see       --
--   if it is null or a duplicate of a previous combination.  A null         --
--   combination is one in which all feature counts are zero.  A duplicate --
--   combination is one in which each feature count 'matches' the           --
--   corresponding feature count of another combination.  In this context, --
--   a 'match' is when both feature counts are zero or both are non-zero.  --
-- ALGORITHM:                                                                 --
--   check for null combination and count non-zero feature counts            --
--   store the feature count total for this combination                      --
--   if this is null combination, increment null counter                     --
--   if not first combination and not null, check for duplicate              --
--   if it is duplicate                                                       --
--       increment duplicate count for matching combination                  --
--       increment total duplicate counter                                    --
--       zero out current combination row of matrix                           --
--   if combination was null or duplicate, decrement combination pointer   --
-- PASSED VARIABLES:  None.                                                   --
-- RETURNS:  None.                                                            --
-- GLOBAL VARIABLES USED:                                                     --
--   Matrix_Pkg.Combination_Matrix                                            --
--   Matrix_Pkg.Current_Comb                                                  --
--   Matrix_Pkg.Null_Count                                                    --
--   Matrix_Pkg.Duplicate_Count                                               --
--   Matrix_Pkg.Max_Features                                                  --
-- GLOBAL VARIABLES CHANGED:                                                  --
--   Matrix_Pkg.Combination_Matrix                                            --
--   Matrix_Pkg.Current_Comb                                                  --
--   Matrix_Pkg.Null_Count                                                    --
--   Matrix_Pkg.Duplicate_Count                                               --
-- FILES READ:  None.                                                         --
-- FILES WRITTEN:  None.                                                      --
-- HARDWARE INPUT:  None.                                                     --
-- HARDWARE OUTPUT:  None.                                                    --
-- MODULES CALLED:  None.                                                     --
-- CALLING MODULES:  ALIANT_Driver                                            --
--                                                                            --
-- AUTHOR:  Capt James S. Marr                                                --
-- HISTORY:  None.                                                            --
--                                                                            --
-- ORDER-OF ANALYSIS:  O(n**2) since the procedure is dominated by two       --
--   nested loops for checking duplicate combinations.  The value for n is --
--   dependent on the number of combinations already in the table and the  --
--    number of features that must be compared before a non-duplicate is    --
--    discovered.  In the best case, the first feature may mismatch; but    --
--    in the worst case the last possible combination is being checked      --
--    against a full combination matrix in which there is no duplicate.     --
--    In this worst case, the actual order-of is:                           --
--    O(Max_Combinations * Max_Features).                                    --
--                                                                           --
-------------------------------------------------------------------------------

  procedure End_Combination is

     Null_Comb      : boolean := true;
     Duplicate_Comb : boolean := true;
     Feature_Count  : natural := 0;

  begin

     -- COUNT THE NUMBER OF NON-ZERO FEATURE COUNTS AND --
     -- SET THE NULL_COMB FLAG TO FALSE IF AT LEAST ONE --
     -- FEATURE COUNT IS NON-ZERO.                      --

     for J in 1..Max_Features loop
         if (Combination_Matrix (Current_Comb,J) /= 0) then
             Null_Comb := false;
             Feature_Count := Feature_Count + 1;
         end if;
     end loop;

     -- STORE THE NUMBER OF FEATURES COUNTED FOR THIS COMBINATION --

     Combination_Matrix (Current_Comb, -1) := Feature_Count;

     -- INCREMENT THE NULL_COMB COUNTER IF NECESSARY --

     if Null_Comb then
         Null_Count := Null_Count + 1;
     end if;

     -- DETERMINE IF THIS COMBINATION IS A DUPLICATE --

     if ((Current_Comb > 1) and (not Null_Comb)) then
         for I in 1..(Current_Comb - 1) loop
             Duplicate_Comb := true;

             -- COMPARE FEATURE COUNTS, ONE BY ONE, BETWEEN   --
             -- CURRENT COMBINATION AND ANOTHER COMBINATION   --
             -- UNTIL ONE OR THE OTHER (NOT BOTH) IS ZERO.    --

             for J in 1..Max_Features loop
                 if (((Combination_Matrix (Current_Comb,J)
                      = 0) and (Combination_Matrix (I,J)
```

```
                  /= 0)) or
                ((Combination_Matrix (Current_Comb,J)
                  /= 0) and (Combination_Matrix (I,J)
                  = 0))) then
                 Duplicate_Comb := false;
                 exit;
             end if;
         end loop;

         -- IF CURRENT COMBINATION IS A DUPLICATE, INCREMENT THE --
         -- DUPLICATE COUNTER OF THE OTHER COMBINATION AND EXIT  --
         -- ENTIRE DUPLICATE CHECKING LOOP.                      --

         if Duplicate_Comb then
             Combination_Matrix (I,0) := Combination_Matrix (I,0) + 1;
             exit;
         end if;
     end loop;
 else

     -- IF CURRENT COMBINATION IS THE FIRST     --
     -- ONE OR IS NULL, IT CAN'T BE A DUPLICATE --

     Duplicate_Comb := false;
 end if;

 -- IF CURRENT COMBINATION IS A DUPLICATE, --
 -- INCREMENT DUPLICATE COUNT AND ZERO OUT --
 -- ALL POSITIONS OF THIS MATRIX ROW.      --

 if Duplicate_Comb then
     Duplicate_Count := Duplicate_Count + 1;
     for J in -1..Max_Features loop
         Combination_Matrix (Current_Comb,J) := 0;
     end loop;
 end if;

 -- IF CURRENT COMBINATION IS DUPLICATE --
 -- OR NULL, DECREMENT CURRENT_COMB.    --

 if (Null_Comb or Duplicate_Comb) then
     Next_Comb := Current_Comb;
     Current_Comb := Current_Comb - 1;
 end if;

end End_Combination;

-------------------------------------------------------------------------------
--                             MODULE HEADER                                 --
--                                                                           --
-- DATE:  31 Aug 90                                                          --
-- VERSION: 1.0                                                        --
-- NAME: Display_Matrix                                                --
-- DESCRIPTION: This procedure is used to display selected combinations. --
--   The combinations are selected based on two threshold values: a dupli- --
--   cation threshold and a feature threshold. The duplication threshold --
--   will select any combinations that have a duplicate count greater or --
--   equal to the specified threshold value. The feature threshold will  --
--   select any combinations that have the same or less number of features --
--   as the threshold value. The selected combinations are displayed to  --
--   show the features included in each combination.                     --
-- ALGORITHM:                                                           --
--   display ALIANT Processing Statistics                               --
--   determine if user would like to select combinations                --
--   while selection is desired                                         --
--       get the threshold values                                       --
--       display the number of selected combinations                    --
--       determine if user would like to display combinations           --
--       if display is desired, display combinations                    --
--       determine if more selection is desired                         --
-- PASSED VARIABLES: None.                                              --
-- RETURNS: None.                                                       --
-- GLOBAL VARIABLES USED:                                               --
--   Matrix_Pkg.Number_Comb_Processed                                   --
--   Matrix_Pkg.Null_Count                                              --
--   Matrix_Pkg.Duplicate_Count                                         --
--   Matrix_Pkg.Current_Comb                                            --
--   Matrix_Pkg.Combination_Matrix                                      --
--   Matrix_Pkg.Max_Features                                            --
-- GLOBAL VARIABLES CHANGED: None.                                      --
-- FILES READ: None.                                                    --
-- FILES WRITTEN: None.                                                 --
-- HARDWARE INPUT: Keyboard.                                            --
-- HARDWARE OUTPUT: CRT.                                                --
-- MODULES CALLED:                                                      --
--   Display_Matrix.Get_User_Input                                      --
--   Text_IO.new_line                                                   --
--   Text_IO.put_line                                                   --
--   Text_IO.new_page                                                   --
--   Text_IO.put                                                        --
--   Matrix_Pkg.Natural_IO.get (instantiation of Text_IO.integer_io)    --
--   Text_IO.skip_line                                                  --
--   Parameter_Pkg.Screen_Delay                                         --
--   Display_Matrix.Check_Paging                                        --
--   Features_Pkg.Get_Feature                                           --
-- CALLING MODULES: ALIANT_Driver.ALIANT_Wrapup                         --
--                                                                      --
-- AUTHOR: Capt James S. Marr                                           --
-- HISTORY: None.                                                       --
--                                                                      --
-- ORDER-OF ANALYSIS: O(n**3) since the procedure is dominated by three --
--   nested loops. The dominant portion is when the user chooses        --
--    threshold values that result in all combinations being displayed.    --
--    In the worst case, the actual order-of is:                           --
--    O(g * Max_Combinations * Max_Features); where g represents the number --
--    of times the user decides to input new threshold values.             --
--                                                                          --
-----------------------------------------------------------------------------

    procedure Display_Matrix is

        Unique_Comb          : Comb_Number_Type := 0;
        Last                 : natural;
        Duplicate_Threshold  : natural;
        Feature_Threshold    : natural;
        Paging_Counter       : natural;
        Combination_Count    : natural;
        User_Input           : string (1..50);
        Paging_Selected      : boolean;
        Quit_Paging          : boolean;
        subtype Option_Type is integer range 1..3;

-----------------------------------------------------------------------------
--                            MODULE HEADER                                 --
--                                                                          --
--  DATE:  31 Aug 90                                                        --
--  VERSION:  1.0                                                           --
--  NAME:  Get_User_Input                                                   --
--  DESCRIPTION:  This is a local procedure to Display_Matrix which was     --
--    created to avoid duplication of code.  The overall purpose is to      --
--    display a selected prompt and then get the user response.             --
--  ALGORITHM:                                                              --
--    display prompt based on option parameter                              --
--    get the user input                                                    --
--  PASSED VARIABLES:  Option                                               --
--  RETURNS:  None.                                                         --
--  GLOBAL VARIABLES USED:                                                  --
--    Display_Matrix.User_Input                                             --
--    Display_Matrix.Last                                                   --
--  GLOBAL VARIABLES CHANGED:                                               --
--    Display_Matrix.User_Input                                             --
--    Display_Matrix.Last                                                   --
--  FILES READ:  None.                                                      --
--  FILES WRITTEN:  None.                                                   --
--  HARDWARE INPUT:  Keyboard.                                              --
--  HARDWARE OUTPUT:  CRT.                                                  --
--  MODULES CALLED:                                                         --
--    Text_IO.new_line                                                      --
--    Text_IO.put                                                           --
--    Text_IO.put_line                                                      --
--    Text_IO.get_line                                                      --
--    Text_IO.skip_line                                                     --
--  CALLING MODULES:                                                        --
--    Matrix_Pkg.Display_Matrix                                           --
--    Matrix_Pkg.Display_Matrix.Check_Paging                              --
--                                                                        --
--  AUTHOR:  Capt James S. Marr                                           --
--  HISTORY:  None.                                                       --
--                                                                        --
--  ORDER-OF ANALYSIS:  O(1) since only sequential statements are used.   --
--                                                                        --
----------------------------------------------------------------------------

        procedure Get_User_Input (Option : Option_Type) is

        begin

            -- DISPLAY APPROPRIATE MESSAGE BASED ON OPTION --

            text_io.new_line;
            case Option is
                when 1 =>
                    text_io.put (" Enter 'y' to " &
                      "SELECT combinations for display >> ");
                when 2 =>
                    text_io.put_line (" Enter 'y' to DISPLAY selected " &
                      "combinations,");
                    text_io.put (" or 'p' to DISPLAY with paging >> ");
                when 3 =>
                    text_io.put_line (" Press RETURN to continue paging,");
                    text_io.put (" or enter 'q' to quit paging >> ");
            end case;

            -- GET USER INPUT FROM KEYBOARD --

            User_Input (1) := ' ';
            text_io.get_line (User_Input, Last);
            if (Last > 49) then
                text_io.skip_line;
            end if;

        exception
            when text_io.end_error =>
                text_io.new_line (2);
                text_io.put_line (" <Matrix_Pkg.Display_Matrix." &
                  "Get_User_Input>");
                text_io.put_line (" *** END-OF-FILE REACHED ON STD INPUT. ***");
                text_io.put_line (" *** PROBABLY INVALID ENTRIES IN THE   ***");
                text_io.put_line (" *** ALIANT BATCH INPUT FILE (IF USED).***");
                text_io.new_line;
                raise Parameter_Pkg.Fatal_Exception;

        end Get_User_Input;
```

```
-----------------------------------------------------------------------------
--                              MODULE HEADER                              --
--                                                                         --
-- DATE:  31 Aug 90                                                        --
-- VERSION:  1.0                                                           --
-- NAME:  Check_Paging                                                     --
-- DESCRIPTION:  This is a local procedure to Display_Matrix.  It is used --
--   to control the paging of CRT output.                                  --
-- ALGORITHM:                                                              --
--   if paging is selected                                                 --
--       increment the page counter                                        --
--       if the screen is full                                             --
--         reset the page counter                                          --
--         determine if user wants to quit paging                          --
--         if user wants to quit paging, set flag                          --
-- PASSED VARIABLES:  None.                                                --
-- RETURNS:  None.                                                         --
-- GLOBAL VARIABLES USED:                                                  --
--   Display_Matrix.Paging_Selected                                        --
--   Display_Matrix.Paging_Counter                                         --
--   Display_Matrix.User_Input                                             --
-- GLOBAL VARIABLES CHANGED:                                               --
--   Display_Matrix.Paging_Counter                                         --
--   Display_Matrix.User_Input                                             --
--   Display_Matrix.Quit_Paging                                            --
-- FILES READ:  None.                                                      --
-- FILES WRITTEN:  None.                                                   --
-- HARDWARE INPUT:  Keyboard.                                              --
-- HARDWARE OUTPUT:  CRT.                                                  --
-- MODULES CALLED:                                                         --
--   Display_Matrix.Get_User_Input                                         --
--   Text_IO.new_line                                                      --
-- CALLING MODULES:                                                        --
--   Matrix_Pkg.Display_Matrix                                             --
--                                                                         --
-- AUTHOR:  Capt James S. Marr                                             --
-- HISTORY:  None.                                                         --
--                                                                         --
-- ORDER-OF ANALYSIS:  O(1) since only sequential statements are used.     --
--                                                                         --
-----------------------------------------------------------------------------

        procedure Check_Paging is

        begin

            -- IF PAGING IS BEING USED, INCREMENT PAGE COUNTER --
            -- AND CHECK FOR END-OF-PAGE.  IF END-OF-PAGE,     --
            -- THEN DETERMINE IF CONTINUED PAGING DESIRED.     --

            if Paging_Selected then
            Paging_Counter := Paging_Counter + 1;
            if Paging_Counter >= 21 then
                Paging_Counter := 0;
                Get_User_Input (3);
                if ((User_Input (1) = 'q') or (User_Input (1) = 'Q')) then
                    Quit_Paging := true;
                end if;
                text_io.new_line;
            end if;
        end if;

    end Check_Paging;

begin

    -- DISPLAY ALIANT PROCESSING STATISTICS --

    text_io.new_line;
    text_io.put_line (" ***** ALIANT Processing Statistics *****");
    text_io.put_line (" Number of combinations processed: " &
                      integer'image (Number_Comb_Processed));
    text_io.put_line (" Null combination count: " &
                      integer'image (Null_Count));
    text_io.put_line (" Duplicate combination count: " &
                      integer'image (Duplicate_Count));
    text_io.put_line (" Resulting combinations: " &
                      integer'image (Current_Comb));
    text_io.put_line (" ****************************************");
    text_io.put_line (" " & ascii.bel);

    -- DETERMINE IF USER WANTS TO SELECT COMBINATIONS FOR DISPLAY --

    Get_User_Input (1);
    while ((User_Input(1) = 'y') or (User_Input(1) = 'Y')) loop

        -- PROMPT USER FOR DUPLICATION_THRESHOLD --
        -- UNTIL VALID VALUE IS PROVIDED.        --

        loop
             begin
                text_io.new_page;
                text_io.new_line;
                text_io.put(" Enter duplication threshold >> ");
                Natural_IO.get (Duplicate_Threshold);
                text_io.skip_line;
                exit;

            exception
                when text_io.data_error =>
                    text_io.new_line;
                    text_io.put_line (" ** INVALID THRESHOLD VALUE -- " &
                "MUST BE A NATURAL NUMBER **");
              text_io.skip_line;
              Parameter_Pkg.Screen_Delay;
          when text_io.end_error =>
              text_io.new_line (2);
              text_io.put_line (" <Matrix_Pkg.Display_Matrix.1>");
              text_io.put_line (" *** END-OF-FILE REACHED ON STD" &
                " INPUT. ***");
              text_io.put_line (" *** PROBABLY INVALID ENTRIES" &
                " IN THE   ***");
              text_io.put_line (" *** ALIANT BATCH INPUT FILE" &
                " (IF USED).***");
              text_io.new_line;
              raise Parameter_Pkg.Fatal_Exception;

      end;
  end loop;

  -- PROMPT USER FOR FEATURE_THRESHOLD --
  -- UNTIL VALID VALUE IS PROVIDED.    --

  loop
      begin
          text_io.new_page;
          text_io.new_line;
          text_io.put(" Enter feature threshold >> ");
          Natural_IO.get (Feature_Threshold);
          text_io.skip_line;
          exit;

      exception
          when text_io.data_error =>
              text_io.new_line;
              text_io.put_line (" ** INVALID THRESHOLD VALUE -- " &
                "MUST BE A NATURAL NUMBER **");
              text_io.skip_line;
              Parameter_Pkg.Screen_Delay;
          when text_io.end_error =>
              text_io.new_line (2);
              text_io.put_line (" <Matrix_Pkg.Display_Matrix.2>");
              text_io.put_line (" *** END-OF-FILE REACHED ON STD" &
                " INPUT. ***");
              text_io.put_line (" *** PROBABLY INVALID ENTRIES" &
                " IN THE   ***");
              text_io.put_line (" *** ALIANT BATCH INPUT FILE" &
                " (IF USED).***");
              text_io.new_line;
              raise Parameter_Pkg.Fatal_Exception;
      end;
  end loop;
```

```
-- COUNT THE NUMBER OF COMBINATIONS THAT SATISFY THE   --
-- DESIRED DUPLICATE AND FEATURE THRESHOLD CONSTRAINTS --

Combination_Count := 0;
for I in 1..Current_Comb loop
    if ((Combination_Matrix (I,0) >= Duplicate_Threshold) and
        (Combination_Matrix (I,-1) <= Feature_Threshold)) then
        Combination_Count := Combination_Count + 1;
    end if;
end loop;

-- DISPLAY THE NUMBER OF COMBINATIONS COUNTED --

text_io.new_page;
text_io.new_line;
if (Combination_Count = 1) then
    text_io.put_line (" There is 1 combination with a " &
      "duplication count >= " &
      integer'image (Duplicate_Threshold) & ",");
    text_io.put_line (" and a feature count <= " &
      integer'image (Feature_Threshold) & ".");
else
    text_io.put_line (" There are " &
      integer'image (Combination_Count) &
      " combinations with a duplication count >= " &
      integer'image (Duplicate_Threshold) & ",");
    text_io.put_line (" and a feature count <= " &
      integer'image (Feature_Threshold) & ".");
end if;

-- IF THERE IS AT LEAST ONE COMBINATION SELECTED, --
-- PROMPT THE USER FOR DISPLAY AND PAGING OPTIONS --

if (Combination_Count /= 0) then
    Get_User_Input (2);
    Paging_Selected := false;
    Quit_Paging := false;
    if ((User_Input(1) = 'y') or (User_Input(1) = 'Y') or
        (User_Input(1) = 'p') or (User_Input(1) = 'P')) then
        if ((User_Input(1) = 'p') or (User_Input(1) = 'P')) then
            Paging_Selected := true;
            Paging_Counter := 0;
        end if;
        text_io.new_page;

        -- DISPLAY SELECTED COMBINATIONS UNTIL END --
        -- OR UNTIL PAGING OPTION IS TERMINATED.   --

        Paging_Loop:
        for I in 1..Current_Comb loop
```

```
                    -- IF COMBINATION MEETS THRESHOLD CONSTRAINTS,   --
                    -- DISPLAY COMBINATION INFO FOLLOWED BY FEATURES --

                    if ((Combination_Matrix (I,0) >=
                      Duplicate_Threshold) and
                      (Combination_Matrix (I,-1) <=
                      Feature_Threshold)) then
                        text_io.new_line;
                        Check_Paging;
                        exit Paging_Loop when Quit_Paging;
                        text_io.put_line (" Combination " &
                          integer'image (I) & ": ");
                        Check_Paging;
                        exit Paging_Loop when Quit_Paging;
                        text_io.put_line (" (" &
                          integer'image (Combination_Matrix (I,0)) &
                          " duplicate[s] )");
                        Check_Paging;
                        exit Paging_Loop when Quit_Paging;
                        text_io.new_line;
                        Check_Paging;
                        exit Paging_Loop when Quit_Paging;

                        -- FOR EACH FEATURE IN SELECTED COMBINATION, --
                        -- DISPLAY FEATURE FOLLOWED BY FEATURE COUNT --

                        for J in 1..Max_Features loop
                            if (Combination_Matrix (I,J) /= 0) then
                                text_io.put_line (" " &
                                  Features_Pkg.Get_Feature (J) &
                                  " (count:" &
                                  integer'image (Combination_Matrix (I,J))
                                  & ")");
                                Check_Paging;
                                exit Paging_Loop when Quit_Paging;
                            end if;
                        end loop;
                    end if;
                end loop Paging_Loop;
            end if;
        end if;
        Get_User_Input (1);
    end loop;

end Display_Matrix;

-------------------------------------------------------------------------------
--                              MODULE HEADER                               --
--                                                                          --
--  DATE:  31 Aug 90                                                        --
--  VERSION:  1.0                                                           --
-- NAME:  Load_Database                                                  --
-- DESCRIPTION:  This procedure loads the AFIS database according to user --
--   specified threshold values.  The threshold values are used in the    --
--   same manner as in Matrix_Pkg.Display_Matrix.                         --
-- ALGORITHM:                                                             --
--   determine if user wants to load database                             --
--   if database load is desired                                          --
--       get the duplicate and feature threshold values                   --
--       display 'loading database' message                               --
--       load selected combinations to the AFIS database                  --
--       display 'loading complete' message                               --
-- PASSED VARIABLES:  None.                                               --
-- RETURNS:  None.                                                        --
-- GLOBAL VARIABLES USED:                                                 --
--   Parameter_Pkg.Database_Filename                                      --
--   Matrix_Pkg.Current_Comb                                              --
--   Matrix_Pkg.Combination_Matrix                                        --
-- GLOBAL VARIABLES CHANGED:  None.                                       --
-- FILES READ:  None.                                                     --
-- FILES WRITTEN:  AFIS Database.                                         --
-- HARDWARE INPUT.  Keyboard.                                             --
-- HARDWARE OUTPUT: CRT, File output.                                     --
-- MODULES CALLED:                                                        --
--   Text_IO.new_page                                                     --
--   Text_IO.new_line                                                     --
--   Text_IO.put                                                          --
--   Text_IO.get_line                                                     --
--   Text_IO.skip_line                                                    --
--   Matrix_Pkg.Natural_IO.get (instantiation of Text_IO.integer_io)      --
--   Parameter_Pkg.Screen_Delay                                           --
--   Text_IO.create                                                       --
--   Text_IO.put_line                                                     --
--   Text_IO.close                                                        --
-- CALLING MODULES:  ALIANT_Driver.ALIANT_Wrapup                          --
--                                                                        --
-- AUTHOR:  Capt James S. Marr                                            --
-- HISTORY:  None.                                                        --
--                                                                        --
-- ORDER-OF ANALYSIS:  O(n**2) since the procedure is dominated by two    --
--   nested loops.  In the worst case when all combinations are written   --
--   to the database, the actual order-of could be as high as:            --
--   O(Max_Combinations * Max_Features).                                  --
--                                                                        --
--------------------------------------------------------------------------

procedure Load_Database is

    Database_File       : text_io.file_type;
    Last                : natural;
    Duplicate_Threshold : natural;
    Feature_Threshold   : natural;
    Combination_Count    : natural := 0;
    User_Input           : string (1..50);

begin

    -- DETERMINE IF USER WANTS TO LOAD AFIS DATABASE --

    text_io.new_page;
    text_io.new_line;
    text_io.put (" Enter 'y' to load AFIS database >> ");
    User_Input (1) := ' ';
    text_io.get_line (User_Input, Last);
    if (Last > 49) then
        text_io.skip_line;
    end if;

    if ((User_Input (1) = 'y') or (User_Input (1) = 'Y')) then

        -- PROMPT USER FOR DUPLICATION_THRESHOLD --
        -- UNTIL VALID VALUE IS PROVIDED.        --

        loop
            begin
                text_io.new_page;
                text_io.new_line;
                text_io.put(" Enter duplication threshold" &
                  " for database >> ");
                Natural_IO.get (Duplicate_Threshold);
                text_io.skip_line;
                exit;

            exception
                when text_io.data_error =>
                    text_io.new_line;
                    text_io.put_line (" ** INVALID THRESHOLD VALUE -- " &
                                      "MUST BE A NATURAL NUMBER **");
                    text_io.skip_line;
                    Parameter_Pkg.Screen_Delay;
            end;

        end loop;

        -- PROMPT USER FOR FEATURE_THRESHOLD --
        -- UNTIL VALID VALUE IS PROVIDED.    --

        loop
            begin
                text_io.new_page;
                text_io.new_line;
                text_io.put(" Enter feature threshold for database >> ");
                Natural_IO.get (Feature_Threshold);
            text_io.skip_line;
            exit;

        exception
            when text_io.data_error =>
                text_io.new_line;
                text_io.put_line (" ** INVALID THRESHOLD VALUE -- " &
                                  "MUST BE A NATURAL NUMBER **");
                text_io.skip_line;
                Parameter_Pkg.Screen_Delay;
        end;

    end loop;

    -- DISPLAY 'LOADING DATABASE' MESSAGE --

    text_io.new_page;
    text_io.new_line;
    text_io.put_line (" Loading AFIS database...");

    -- CREATE DATABASE FILE AND INITIALIZE FILE HEADER --

    text_io.create (Database_File, text_io.out_file,
                    Parameter_Pkg.Database_Filename);
    text_io.put_line (Database_File, ">>>> AFIS DATABASE FILE <<<<");

    -- SEARCH COMBINATIONS FOR DESIRED SELECTION --

    for I in 1..Current_Comb loop
        if ((Combination_Matrix (I,0) >= Duplicate_Threshold) and
            (Combination_Matrix (I,-1) <= Feature_Threshold)) then
            text_io.new_line (Database_File);

            -- FOR EACH SELECTED COMBINATION, OUTPUT A UNIQUE --
            -- RECORD IDENTIFIER (USE COMBINATION NUMBER).    --

            text_io.put (Database_File, integer'image (I) & ": ");

            -- FOR EACH FEATURE OF SELECTED COMBINATIONS,    --
            -- OUTPUT A '1' IS THE FEATURE COUNT IS NON-ZERO --
            -- OR A '0' IS THE FEATURE COUNT IS ZERO.        --

            for J in 1..Max_Features loop
                if (Combination_Matrix (I,J) = 0) then
                    text_io.put (Database_File, integer'image (0));
                else
                    text_io.put (Database_File, integer'image (1));
                end if;
            end loop;
            Combination_.ount := Combination_Count + 1;
        end if;
```

```
            end loop;

            -- CLOSE THE DATABASE FILE AND DISPLAY COMPLETION MESSAGE --

            text_io.close (Database_File);
            text_io.new_line;
            text_io.put (" ...The AFIS database has been loaded with " &
              integer'image (Combination_Count) & " record");
            if (Combination_Count = 1) then
                text_io.put_line (".");
            else
                text_io.put_line ("s.");
            end if;
        end if;

    exception
        when text_io.name_error =>
            text_io.new_line;
            text_io.put_line (" <Matrix_Pkg.Load_Database>");
            text_io.put_line (" *** NAME EXCEPTION ERROR RAISED WHILE ***");
            text_io.put_line (" *** TRYING TO CREATE DATABASE FILE.   ***");
            text_io.put_line (" *** CHECK FILENAME IN PARAMETER       ***");
            text_io.put_line (" *** PACKAGE FOR PROPER FORMAT.        ***");
            text_io.new_line;
            raise Parameter_Pkg.Fatal_Exception;
        when text_io.end_error =>
            text_io.new_line (2);
            text_io.put_line (" <Matrix_Pkg.Load_Database>");
            text_io.put_line (" *** END-OF-FILE REACHED ON STD INPUT. ***");
            text_io.put_line (" *** PROBABLY INVALID ENTRIES IN THE   ***");
            text_io.put_line (" *** ALIANT BATCH INPUT FILE (IF USED).***");
            text_io.new_line;
            raise Parameter_Pkg.Fatal_Exception;
        when others =>
            text_io.new_line;
            text_io.put_line (" <Matrix_Pkg.Load_Database>");
            text_io.put_line (" *** UNKNOWN EXCEPTION RAISED ***");
            text_io.put_line (" *** WHILE LOADING DATABASE.  ***");
            text_io.new_line;
            text_io.close (Database_File);
            raise Parameter_Pkg.Fatal_Exception;

    end Load_Database;

end Matrix_Pkg;

-------------------------------------------------------------------------------
--                           MAIN DRIVER HEADER                              --
--                                                                           --
-- DATE:  31 Aug 90                                                          --
-- VERSION:  1.0                                                             --
-- NAME:  ALIANT_Driver                                                  --
-- DESCRIPTION:  This procedure is the main driver for the ALIANT        --
--   prototype.  The ALIANT prototype reads in combinations from Gen     --
--   via an ASCII file.  Actually, the entire ALIANT prototype includes  --
--   the Gen software and the UNIX script file that ties the Gen and     --
--   Ada code together.  This driver is responsible for calling the      --
--   various procedures that analyze the Ada feature combinations.       --
-- ALGORITHM:                                                            --
--       call Matrix_Pkg.Initialize_Matrix                               --
--       call Features_Pkg.Load_Features_Table                           --
--       call Lex_Pkg.Opengen                                            --
--       call Lex_Pkg.Yylex (return Token)                               --
--       while (not end of Gen input file) loop                          --
--           case Token is                                               --
--               when feature token =>                                   --
--                 call Matrix_Pkg.Count_Feature (send Token)            --
--               when start token =>                                     --
--                 call Matrix_Pkg.Start_Combination                     --
--               when end token =>                                       --
--                 call Matrix_Pkg.End_Combination                       --
--               when others =>                                          --
--                 display error message                                 --
--           end case                                                    --
--           call Lex_Pkg.Yylex (return Token)                           --
--       end while loop                                                  --
--       call ALIANT_Wrapup                                              --
-- PASSED VARIABLES:  None.                                              --
-- RETURNS:  None.                                                       --
-- GLOBAL VARIABLES USED:  None.                                         --
-- GLOBAL VARIABLES CHANGED:  None.                                      --
-- FILES READ:  None.                                                    --
-- FILES WRITTEN:  None.                                                 --
-- HARDWARE INPUT:  None.                                                --
-- HARDWARE OUTPUT:  CRT.                                                --
-- MODULES CALLED:                                                       --
--   Matrix_Pkg.Initialize_Matrix                                        --
--   Features_Pkg.Load_Features_Table                                    --
--   Lex_Pkg.Opengen                                                     --
--   Text_IO.new_page                                                    --
--   Text_IO.new_line                                                    --
--   Text_IO.put_line                                                    --
--   Lex_Pkg.Yylex                                                       --
--   Matrix_Pkg.Count_Feature                                            --
--   Matrix_Pkg.Start_Combination                                        --
--   Matrix_Pkg.End_Combination                                          --
--   ALIANT_Driver.ALIANT_Wrapup                                         --
-- REQUIRED LIBRARY UNITS:                                               --
--   Matrix_Pkg                                                          --
--   Features_Pkg                                                        --
--   Parameter_Pkg                                                       --
--   Text_IO                                                             --
--    Lex_Pkg                                                              --
--                                                                         --
-- AUTHOR:  Capt James S. Marr                                             --
-- HISTORY:  None.                                                         --
--                                                                         --
-- ORDER-OF ANALYSIS:  O(n**3) since t!   procedure is dominated by the    --
--    order-of for ALIANT_Wrapup.                                          --
--                                                                         --
-----------------------------------------------------------------------------


with Matrix_Pkg;
with Features_Pkg;
with Parameter_Pkg;
with text_io;
with Lex_Pkg;
procedure ALIANT_Driver is

    Token        : natural;
    Max_Features : constant Parameter_Pkg.Parameter_Type
      := Parameter_Pkg.Get_Max_Features;

-----------------------------------------------------------------------------
--                             MODULE HEADER                               --
--                                                                         --
-- DATE:  31 Aug 90                                                        --
-- VERSION:  1.0                                                           --
-- NAME:  ALIANT_Wrapup                                                    --
-- DESCRIPTION:  This procedure is local to the ALIANT_Driver and was     --
--    created to eliminate duplication of code.  It contains those actions --
--    that take place after combination processing is completed.          --
-- ALGORITHM:                                                              --
--    close the Gen input file                                             --
--    call Matrix_Pkg.Display_Matrix                                       --
--    call Matrix_Pkg.Load_Database                                        --
--    display a normal termination message                                 --
-- PASSED VARIABLES:  None.                                                --
-- RETURNS:  None.                                                         --
-- GLOBAL VARIABLES USED:  None.                                           --
-- GLOBAL VARIABLES CHANGED:  None.                                        --
-- FILES READ:  None.                                                      --
-- FILES WRITTEN:  None.                                                   --
-- HARDWARE INPUT:  None.                                                  --
-- HARDWARE OUTPUT:  CRT.                                                  --
-- MODULES CALLED:                                                         --
--    Text_IO.new_line                                                     --
--    Lex_Pkg.Closegen                                                     --
--    Matrix_Pkg.Display_Matrix                                            --
--    Matrix_Pkg.Load_Database                                             --
--    Text_IO.put_line                                                     --
-- CALLING MODULES:  ALIANT_Driver                                         --
--                                                                            --
--  AUTHOR:  Capt James S. Marr                                               --
--  HISTORY:  None.                                                           --
--                                                                            --
--  ORDER-OF ANALYSIS:  O(n**3) since this procedure is dominated by the     --
--    order-of for Matrix_Pkg.Display_Matrix.                                 --
--                                                                            --
--------------------------------------------------------------------------------

    procedure ALIANT_Wrapup is

    begin

        -- CLOSE THE GEN INPUT FILE,              --
        -- CALL DISPLAY_MATRIX AND LOAD_DATABASE,  --
        -- THEN DISPLAY NORMAL TERMINATION MESSAGE --

        text_io.new_line;
        Lex_Pkg.Closegen;
        Matrix_Pkg.Display_Matrix;
        Matrix_Pkg.Load_Database;
        text_io.new_line;
        text_io.put_line (" **************************");
        text_io.put_line (" ** Exiting ALIANT driver **");
        text_io.put_line (" **************************");

    end ALIANT_Wrapup;

begin

    -- PERFORM INITIALIZATION PROCEDURES AND OPEN THE GEN INPUT FILE --

    Matrix_Pkg.Initialize_Matrix;
    Features_Pkg.Load_Features_Table;
    Lex_Pkg.Opengen;

    -- DISPLAY PROCESSING MESSAGE --

    text_io.new_page;
    text_io.new_line;
    text_io.put_line (" ************************************************");
    text_io.put_line (" ** Processing Gen combinations, please wait... **");
    text_io.put_line (" ************************************************");

    -- PROCESS EACH TOKEN IN THE GEN INPUT FILE UNTIL END-OF-FILE --

    Token := Lex_Pkg.Yylex;
    while (Token /= 0) loop

        -- IF THE TOKEN IS A FEATURE, CALL COUNT_FEATURE --
        if (Token >= 1) and (Token <= Max_Features) then
            Matrix_Pkg.Count_Feature (Token);

        -- OTHERWISE, TAKE THE APPROPRIATE ACTION --

        else
            case Token is
                when 995      => Matrix_Pkg.Start_Combination;
                when 996      => Matrix_Pkg.End_Combination;
                when 998..999 => null; -- blanks and carriage return
                when others   =>
                  text_io.new_line;
                  text_io.put_line (" ** Undefined Token #" &
                    integer'image (Token) &
                    ", regenerate lex_spec file from input grammar. **");
            end case;
        end if;
        Token := Lex_Pkg.Yylex;
    end loop;
    ALIANT_Wrapup;

exception
    when Parameter_Pkg.Fatal_Exception =>
        text_io.new_line;
        text_io.put_line (" ***********************" &
          "**************************");
        text_io.put_line (" ** Exiting ALIANT driver" &
          " due to fatal exception **");
        text_io.put_line (" ***********************" &
          "**************************");
    when Parameter_Pkg.Partial_Exception => ALIANT_Wrapup;
    when others =>
        text_io.new_line;
        text_io.put_line (" ***************************" &
          "***********************");
        text_io.put_line (" ** Exiting ALIANT driver due " &
          "to unknown exception **");
        text_io.put_line (" ***************************" &
          "***********************");
        text_io.new_line;
        raise;

end ALIANT_Driver;
```

Appendix E. *Input Grammars*

This appendix contains the input grammars used in the ALIANT prototype research. The grammars are in a format compatible with the Gen software, which is used to generate valid "sentences" or programs described by a grammar. The first grammar, Adagen1, was annotated by reducing the right hand side of each Ada production to a literal string in quotes, if it contained any terminal symbols. As a result, most productions are never "reached" during test case generation. They are left in the grammar for consistency and later modification. The second grammar, Adagen2, was annotated using the list of some 297 "primary features" as a guide. There are still many productions that are never reached during test case generation. Some of the Adagen2 productions include alternative symbols with specified randomness percentages.

E.1 Adagen1 Grammar

```
/*************************** ADAGEN1 **********************************/

graphic_character = ( "graphic_character " )

basic_graphic_character = ( "basic_graphic_character " )

basic_character = ( "basic_character " )

identifier = ( "identifier " )

letter_or_digit = ( "letter_or_digit " )

letter = ( "letter " )

numeric_literal = ( decimal_literal | % based_literal )

decimal_literal = ( "decimal_literal " )
```

```
integer = ( "integer " )

exponent = ( "exponent " )

based_literal = ( "based_literal " )

base = ( "base " )

based_integer = ( "based_integer " )

extended_digit = ( "extended_digit " )

character_literal = ( "character_literal " )

string_literal = ( "string_literal " )

pragma = ( "pragma " )

argument_association = ( "argument_association " )

basic_declaration = (
  object_declaration | % number_declaration
  | % type_declaration | % subtype_declaration
  | % subprogram_declaration | % package_declaration
  | % task_declaration | % generic_declaration
  | % exception_declaration | % generic_instantiation
  | % renaming_declaration | % deferred_constant_declaration )

object_declaration = ( "object_declaration " )

number_declaration = ( "number_declaration " )

identifier_list = ( "identifier_list " )

type_declaration = ( full_type_declaration
  | % incomplete_type_declaration | % private_type_declaration )

full_type_declaration = ( "full_type_declaration " )

type_definition = (
  enumeration_type_definition | % integer_type_definition
  | % real_type_definition | % array_type_definition
  | % record_type_definition | % access_type_definition
  | % derived_type_definition )

subtype_declaration = ( "subtype_declaration " )

subtype_indication = ( "subtype_indication " )

type_mark = ( "type_mark " )
```

```
constraint = (
  range_constraint | % floating_point_constraint
  | % fixed_point_constraint | % index_constraint
  | % discriminant_constraint )

derived_type_definition = ( "derived_type_definition " )

range_constraint = ( "range_constraint " )

range = ( "range " )

enumeration_type_definition = ( "enumeration_type_definition " )

enumeration_literal_specification = ( "enumeration_literal_specification " )

enumeration_literal = ( "enumeration_literal " )

integer_type_definition = ( "integer_type_definition " )

real_type_definition = (
  floating_point_constraint | % fixed_point_constraint )

floating_point_constraint = (
  floating_accuracy_definition ( "" | % range_constraint ) )

floating_accuracy_definition = ( "floating_accuracy_definition " )

fixed_point_constraint = (
  fixed_accuracy_definition ( "" | % range_constraint ) )

fixed_accuracy_definition = ( "fixed_accuracy_definition " )

array_type_definition = (
  unconstrained_array_definition | % constrained_array_definition )

unconstrained_array_definition = ( "unconstrained_array_definition " )

constrained_array_definition = ( "constrained_array_definition " )

index_subtype_definition = ( "index_subtype_definition " )

index_constraint = ( "index_constraint " )

discrete_range = ( "discrete_range " )

record_type_definition = ( "record_type_definition " )

component_list = ( "component_list " )

component_declaration = ( "component_declaration " )
```

```
component_subtype_definition = ( "component_subtype_definition " )

discriminant_part = ( "discriminant_part " )

discriminant_specification = ( "discriminant_specification " )

discriminant_constraint = ( "discriminant_constraint " )

discriminant_association = ( "discriminant_association " )

variant_part = ( "variant_part " )

variant = ( "variant " )

choice = ( "choice " )

access_type_definition = ( "access_type_definition " )

incomplete_type_declaration = ( "incomplete_type_declaration " )

declarative_part = (
  ( "" | % basic_declarati ._item more_basic_decl )
  ( "" | % later_declarative_item more_later_decl ) )

/***** more_basic_decl and more_later_decl added for Gen *******/

more_basic_decl = ( "" | % basic_declarative_item more_basic_decl )

more_later_decl = ( "" | % later_de:larative_item more_later_decl )

basic_declarative_item = ( basic_declaration
  | % representation_clause | % use_clause )

later_declarative_item = ( body
  | % subprogram_declaration | % package_declaration
  | % task_declaration | % generic_declaration
  | % use_clause | % generic_instantiation )

body = ( proper_body | % body_stub )

proper_body = ( subprogram_body | % package_body | % task_body )

name = ( simple_name
  | % character_literal | % operator_symbol
  | % indexed_component | % slice
  | % selected_component | % attribute )

simple_name = ( identifier )

prefix = ( name | % function_call )
```

```
indexed_component = ( "indexed_component " )

slice = ( "slice " )

selected_component = ( "selected_component " )

selector = ( "selector " )

attribute = ( "attribute " )

attribute_designator = ( "attribute_designator " )

aggregate = ( "aggregate " )

component_association = ( "component_association " )

expression = ( "expression " )

relation = ( "relation " )

simple_expression = ( "simple_expression " )

term = ( "term " )

factor = ( "factor " )

primary = ( "primary " )

logical_operator = ( "logical_operator " )

relational_operator = ( "relational_operator " )

binary_adding_operator = ( "binary_adding_operator " )

unary_adding_operator = ( "unary_adding_operator " )

multiplying_operator = ( "multiplying_operator " )

highest_precedence_operator = ( "highest_precedence_operator " )

type_conversion = ( "type_conversion " )

qualified_expression = ( "qualified_expression " )

allocator = ( "allocator " )

sequence_of_statements = ( statement ( "" | % statement more_statements ) )

/******* more_statements added for Gen *****/

more_statements = ( "" | % statement more_statements )
```

```
statement = (
  ( "" | % label more_labels ) ( simple_statement | % compound_statement ) )

/******* more_labels added for Gen *******/

more_labels = ( "" | % label more_labels )

simple_statement = ( null_statement
  | % assignment_statement | % procedure_call_statement
  | % exit_statement | % return_statement
  | % goto_statement | % entry_call_statement
  | % delay_statement | % abort_statement
  | % raise_statement | % code_statement )

compound_statement = (
  if_statement | % case_statement
  | % loop_statement | % block_statement
  | % accept_statement | % select_statement )

label = ( "label " )

null_statement = ( "null_statement " )

assignment_statement = ( "assignment_statement " )

if_statement = ( "if_statement " )

condition = ( "condition " )

case_statement = ( "case_statement " )

case_statement_alternative = ( "case_statement_alternative " )

loop_statement = ( "loop_statement " )

iteration_scheme = ( "iteration_scheme " )

loop_parameter_specification = ( "loop_parameter_specification " )

block_statement = ( "block_statement " )

exit_statement = ( "exit_statement " )

return_statement = ( "return_statement " )

goto_statement = ( "goto_statement " )

subprogram_declaration = ( subprogram_specification )

subprogram_specification = ( "subprogram_specification " )
```

```
designator = ( identifier | % operator_symbol )

operator_symbol = ( string_literal )

formal_part = ( "formal_part " )

parameter_specification = ( "parameter_specification " )

mode = ( "mode " )

subprogram_body = ( "subprogram_body " )

procedure_call_statement = ( "procedure_call_statement " )

function_call = ( "function_call " )

actual_parameter_part = ( "actual_parameter_part " )

parameter_association = ( "parameter_association " )

formal_parameter = ( "formal_parameter " )

actual_parameter = ( "actual_parameter " )

package_declaration = ( package_specification )

package_specification = ( "package_specification " )

package_body = ( "package_body " )

private_type_declaration = ( "private_type_declaration " )

deferred_constant_declaration = ( "deferred_constant_declaration " )

use_clause = ( "use_clause " )

renaming_declaration = ( "renaming_declaration " )

task_declaration = ( task_specification )

task_specification = ( "task_specification " )

task_body = ( "task_body " )

entry_declaration = ( "entry_declaration " )

entry_call_statement = ( "entry_call_statement " )

accept_statement = ( "accept_statement " )
```

```
entry_index = ( expression )

delay_statement = ( "delay_statement " )

select_statement = ( selective_wait
  | % conditional_entry_call | % timed_entry_call )

selective_wait = ( "selective_wait " )

select_alternative = ( "select_alternative " )

selective_wait_alternative = ( accept_alternative
  | % delay_alternative | % terminate_alternative )

accept_alternative = (
  accept_statement ( "" | % sequence_of_statements ) )

delay_alternative = (
  delay_statement ( "" | % sequence_of_statements ) )

terminate_alternative = ( "terminate_alternative " )

conditional_entry_call = ( "conditional_entry_call " )

timed_entry_cal?   ( "timed_entry_call " )

abort_statement = ( "abort_statement " )

compilation = ( "START_COMPILATION: "
  ( "" | % compilation_unit more_units ) ":END_COMPILATION \n" )

/**** more_units added for Gen *****/

more_units = ( "" | % compilation_unit more_units )

compilation_unit = (
  context_clause library_unit
  | % context_clause secondary_unit )

library_unit = (
  subprogram_declaration | % package_declaration
  | % generic_declaration | % generic_instantiation
  | % subprogram_body )

secondary_unit = ( library_unit_body | % subunit )

library_unit_body = ( subprogram_body | % package_body )

context_clause = (
  "" | % ( with_clause ( "" | % use_clause more_use ) context_clause ) )
```

```
/*******  more_use added for Gen *****/

more_use = ( "" | % use_clause more_use )

with_clause = ( "with_clause " )

body_stub = ( "body_stub " )

subunit = ( "subunit " )

exception_declaration = ( "exception_declaration " )

exception_handler = ( "exception_handler " )

exception_choice = ( "exception_choice " )

raise_statement = ( "raise_statement " )

generic_declaration = ( generic_specification )

generic_specification = (
  generic_formal_part subprogram_specification
  | % generic_formal_part package_specification )

generic_formal_part = ( "generic_formal_part " )

generic_parameter_declaration = ( "generic_parameter_declaration " )

generic_type_definition = ( "generic_type_definition " )

generic_instantiation = ( "generic_instantiation " )

generic_actual_part = ( "generic_actual_part " )

generic_association = ( "generic_association " )

generic_formal_parameter = ( "generic_formal_parameter " )

generic_actual_parameter = ( "generic_actual_parameter " )

representation_clause = (
  type_representation_clause | % address_clause )

type_representation_clause = ( length_clause
  | % enumeration_representation_clause
  | % record_representation_clause )

length_clause = ( "length_clause " )

enumeration_representation_clause = ( "enumeration_representation_clause " )
```

```
record_representation_clause = ( "record_representation_clause " )

alignment_clause = ( "alignment_clause " )

component_clause = ( "component_clause " )

address_clause = ( "address_clause " )

code_statement = ( "code_statement " )
```

E.2 Adagen2 Grammar

```
/**************************** ADAGEN2 **********************************/

graphic_character = ( "graphic_character " )

basic_graphic_character = ( "basic_graphic_character " )

basic_character = ( "basic_character " )

identifier = ( "identifier " )

letter_or_digit = ( "letter_or_digit " )

letter = ( "letter " )

numeric_literal = ( decimal_literal | % based_literal )

decimal_literal = ( "integer_literal " | % "real_literal " )

integer = ( "integer " )

exponent = ( "exponent " )

based_literal = ( "based_literal " )

base = ( "base " )

based_integer = ( "based_integer " )

extended_digit = ( "extended_digit " )

character_literal = ( "character_literal " )

string_literal = ( "string_literal " )
```

```
pragma = ( "pragma " | % "pragma:argument_association " | %
  "predef_pragma " )

argument_association = ( "argument_association " )

basic_declaration = (
  object_declaration | % 14 number_declaration
  | % 3 type_declaration | % 14 subtype_declaration
  | % 14 subprogram_declaration | % 10 package_declaration
  | % 10 task_declaration | % 7 generic_declaration
  | % 8 exception_declaration | % 8 generic_instantiation
  | % 7 renaming_declaration | % 5 deferred_constant_declaration )

object_declaration = ( "object_decl " | % 35 "object_init_val " | % 30
  "object_init_val_constrained_array " | % 25 "constant_decl " )

number_declaration = ( "number_decl " )

identifier_list = ( "identifier_list " )

type_declaration = ( full_type_declaration
  | % 50 incomplete_type_declaration | % 15 private_type_declaration )

full_type_declaration = ( "full_type_decl " ( "" | % discriminant_part ) )

type_definition = (
  enumeration_type_definition | % integer_type_definition
  | % real_type_definition | % array_type_definition
  | % record_type_definition | % access_type_definition
  | % derived_type_definition )

subtype_declaration = ( "subtype_decl " subtype_indication )

subtype_indication = ( "subtype_indic "  ( "" | % constraint ) )

type_mark = ( "type_mark " )

constraint = (
  range_constraint | % floating_point_constraint
  | % fixed_point_constraint | % index_constraint
  | % discriminant_constraint )

derived_type_definition = ( "derived_type_def " )

range_constraint = ( range )

range = ( "range_attribute " | % "explicit_range " )

enumeration_type_definition = ( "enum_type_def " )

enumeration_literal_specification = ( "enumeration_literal_specification " )
```

```
enumeration_literal = ( "enumeration_literal " )

integer_type_definition = ( "integer_type_def " )

real_type_definition = (
  "floating_point_type_def " | % "fixed_point_type_def " )

floating_point_constraint = (
  "floating_point_constraint " ( "" | % range_constraint ) )

floating_accuracy_definition = ( "floating_accuracy_definition " )

fixed_point_constraint = (
  "fixed_point_constraint " ( "" | % range_constraint ) )

fixed_accuracy_definition = ( "fixed_accuracy_definition " )

array_type_definition = ( ( "array_type_def " | % "array_of:access " | %
  "array_of:boolean " | % "array_of:integer " | % "array_of:real " | %
  "array_of:record " | % "array_of:task " )
  ( unconstrained_array_definition | % constrained_array_definition ) )

unconstrained_array_definition = ( "unconstrained_array_def " )

constrained_array_definition = ( "constrained_array_def " )

index_subtype_definition = ( "index_subtype_definition " )

index_constraint = ( "index_constraint " )

discrete_range = ( "discrete_range " )

record_type_definition = ( ( "record_type_def " | % "record_of:access " | %
  "record_of:array " | % "record_of:record " | % "record_of:task " )
   component_list )

component_list = ( component_declaration | % 50 "null_component_list " | % 5
  ( ( "" | % component_declaration ) variant_part ) )

component_declaration = ( "component_decl:default " | %
  "component_decl:no_default " )

component_subtype_definition = ( "component_subtype_definition " )

discriminant_part = ( discriminant_specification )

discriminant_specification = ( "discriminant_spec:default " | %
  "discriminant_spec:no_default " )

discriminant_constraint = ( "discriminant_constraint " )
```

```
discriminant_association = ( "discriminant_association " )

variant_part = ( "variant_part "  ( "" | % variant ) )

variant = ( choice component_list )

choice = ( "variant_choice " | % "variant_choice_others " )

access_type_definition = ( "access_type_def " | % "access_to:array " | %
  "access_to:record " | % "access_to:task " )

incomplete_type_declaration = ( "incomplete_type_decl " ( "" | %
  discriminant_part ) )

declarative_part = (
  ( "" | % basic_declarative_item more_basic_decl )
  ( "" | % later_declarative_item more_later_decl ) )

/*****  more_basic_decl and more_later_decl added for Gen *******/

more_basic_decl = ( "" | % 80 basic_declarative_item more_basic_decl )

more_later_decl = ( "" | % 80 later_declarative_item more_later_decl )

basic_declarative_item = ( basic_declaration
  | % 60 representation_clause | % 20 use_clause )

later_declarative_item = ( body
  | % 20 subprogram_declaration | % 20 package_declaration
  | % 20 task_declaration | % 10 generic_declaration
  | % 13 use_clause | % 4 generic_instantiation )

body = ( proper_body | % 80 body_stub )

proper_body = ( subprogram_body | % 50 package_body | % 30 task_body )

name = ( simple_name
  | % character_literal | % operator_symbol
  | % indexed_component | % slice
  | % selected_component | % attribute )

simple_name = ( identifier )

prefix = ( name | % function_call )

indexed_component = ( "indexed_component " )

slice = ( "slice " )

selected_component = ( "selected_component "  prefix selector )
```

```
selector = ( simple_name | % character_literal | %
  operator_symbol | % "selector_all " )

attribute = ( "attribute " | % "predef_attr " )

attribute_designator = ( "attribute_designator " )

aggregate = ( component_association )

component_association = ( "aggregate " | % "named_component_association " )

expression = ( relation ( "" | % ( ( logical_operator | % 60 "andthen " | % 20
  "orelse ") relation ) ) )

relation = ( simple_expression ( ( relational_operator simple_expression ) | %
  "" | % ( ( ( "membership_test_in " | % "membership_test_not_in " )
  ( range | % type_mark ) ) ) ) )

simple_expression = ( "simple_expression "  ( "" | % unary_adding_operator )
  term ( "" | % ( binary_adding_operator term ) ) )

term = ( factor ( "" | % ( multiplying_operator factor ) ) )

factor = ( ( primary ( "" | % ( "exponentiation " primary ) ) ) | %
  ( "absolute_value " primary ) | % ( "not_operator " primary ) )

primary = ( numeric_literal | % "null_access_value " | % aggregate | %
  string_literal | % name | % allocator | % function_call | %
  type_conversion | % qualified_expression | % "parenthesized_expr " )

logical_operator = ( "and_operator " | % "or_operator " | % "xor_operator " )

relational_operator = ( "equality " | % "inequality " | % "less_than " | %
  "less_than_or_equal_to " | % "greater_than " | %
  "greater_than_or_equal_to " )

binary_adding_operator = ( "addition " | % "subtraction " | % "catenation " )

unary_adding_operator = ( "unary_addition " | % "unary_minus " )

multiplying_operator = ( "multiplication " | % "division " | %
  "mod_operator " | % "rem_operator " )

highest_precedence_operator = ( "exponentiation " | % "absolute_value " | %
  "not_operator " )

type_conversion = ( "type_conversion " )

qualified_expression = ( "qualified_expr " )
```

```
allocator = ( "alloc:qualified_expr " | % "alloc:subtype_indic_constr "
  | % "alloc:subtype_indic_no_constr " )

sequence_of_statements = ( statement ( "" | % statement more_statements ) )

/******* more_statements added for Gen *****/

more_statements = ( "" | % 80 statement more_statements )

statement = (
( "" | % label more_labels ) ( simple_statement | % compound_statement ) )

/******* more_labels added for Gen *******/

more_labels = ( "" | % 80 label more_labels )

simple_statement = ( null_statement
  | % assignment_statement | % procedure_call_statement
  | % exit_statement | % return_statement
  | % goto_statement | % entry_call_statement
  | % delay_statement | % abort_statement
  | % raise_statement | % code_statement )

compound_statement = (
  if_statement | % case_statement
  | % loop_statement | % block_statement
  | % accept_statement | % select_statement )

label = ( "label " )

null_statement = ( "null_statement " )

assignment_statement = ( "assignment_statement " expression )

if_statement = ( "if_statement "  sequence_of_statements )

condition = ( "condition " )

case_statement = ( "case_statement " expression )

case_statement_alternative = ( "case_statement_alternative " )

loop_statement = ( "loop_statement " ( "" | % iteration_scheme )
  sequence_of_statements )

iteration_scheme = ( ( "iteration_scheme:for "
  loop_parameter_specification ) | % ( "iteration_scheme:while "
  condition ) )

loop_parameter_specification = ( "loop_param_spec:up " | %
  "loop_param_spec:down " )
```

```
block_statement = ( "block_statement "  ( "" | % declarative_part )
  sequence_of_statements ( "" | % exception_handler ) )

exit_statement = ( "exit_statement " )

return_statement = ( "return_statement " )

goto_statement = ( "goto_statement " )

subprogram_declaration = ( subprogram_specification )

subprogram_specification = ( ( "subprogram_decl:procedure " | %
  "subprogram_decl:function " ) ( "" | % formal_part ) )

designator = ( identifier | % operator_symbol )

operator_symbol = ( "user_defined_operator " )

formal_part = ( parameter_specification )

parameter_specification = ( "subprog_param_spec:default " | %
  "subprog_param_spec:in " | % "subprog_param_spec:in default " | %
  "subprog_param_spec:in_out " | % "subprog_param_spec:no_default " | %
  "subprog_param_spec:out " )

mode = ( "mode_in " | % "mode_in_default " | %
  "mode_in_out " | % "mode_out " )

subprogram_body = ( ( "procedure_body " | % "function_body " ) ( "" | %
  declarative_part ) sequence_of_statements ( "" | % exception_handler ) )

procedure_call_statement = ( "procedure_call_statement " )

function_call = ( "function_call " )

actual_parameter_part = ( "actual_parameter_part " )

parameter_association = ( "parameter_association " )

formal_parameter = ( "formal_parameter " )

actual_parameter = ( "actual_parameter " )

package_declaration = ( package_specification )

package_specification = ( "package_spec " ( "" | % basic_declarative_item ) )

package_body = ( "package_body " ( "" | % basic_declarative_item )
  sequence_of_statements ( "" | % exception_handler ) )
```

```
private_type_declaration = ( ( "private_type_decl " | %
  "limited_private_type_decl " ) ( "" | % discriminant_part ) )

deferred_constant_declaration = ( "deferred_constant_declaration " )

use_clause = ( "use_clause " )

renaming_declaration = ( "rename:entry " | % "rename:exception " | %
  "rename:object " | % "rename:package " | % "rename:subprog " | %
  "rename:subprog_or_entry " )

task_declaration = ( task_specification )

task_specification = ( ( "task_spec " | % "task_type_spec " )
  ( "" | % entry_declaration ) ( "" | % representation_clause ) )

task_body = ( "task_body "  ( "" | % declarative_part ) sequence_of_statements
  ( "" | % exception_handler ) )

entry_declaration = ( "entry_decl " | % "entry_family_decl "
  ( "entry_param_spec " | % "entry_param_spec:default " | %
  "entry_param_spec:in " | % "entry_param_spec:in_default " | %
  "entry_param_spec:in_out " | % "entry_param_spec:no_default " | %
  "entry_param_spec:out " ) )

entry_call_statement = ( "entry_call_statement " )

accept_statement = ( "accept_statement " )

entry_index = ( expression )

delay_statement = ( "delay_statement " )

select_statement = ( selective_wait
  | % conditional_entry_call | % timed_entry_call )

selective_wait = ( "sel_wait:accept_alt " | % "sel_wait:accept_alt_guarded "
  | % "sel_wait:accept_alt_unguarded " | % "sel_wait:delay_alt " | %
  "sel_wait:delay_alt_guarded " | % "sel_wait:delay_alt_unguarded " | %
  "sel_wait:else_part " | % "sel_wait:term_alt " | %
  "sel_wait:term_alt_guarded " | % "sel_wait:term_alt_unguarded " )

select_alternative = ( "select_alternative " )

selective_wait_alternative = ( accept_alternative
  | % delay_alternative | % terminate_alternative )

accept_alternative = (
  accept_statement ( "" | % sequence_of_statements ) )

delay_alternative = (
  delay_statement ( "" | % sequence_of_statements ) )

terminate_alternative = ( "terminate_alternative " )

conditional_entry_call = ( "conditional_entry_call " entry_call_statement
  sequence_of_statements )

timed_entry_call = ( "timed_entry_call "  entry_call_statement
  delay_alternative )

abort_statement = ( "abort_statement " )

compilation = ( "START_COMPILATION: "
  ( "" | % 0 compilation_unit more_units ) ":END_COMPILATION \n" )

/**** more_units added for Gen *****/

more_units = ( "" | % 90 compilation_unit more_units )

compilation_unit = (
  context_clause library_unit
  | % context_clause secondary_unit )

library_unit = (
  subprogram_declaration | % package_declaration
  | % generic_declaration | % generic_instantiation
  | % subprogram_body )

secondary_unit = ( library_unit_body | % subunit )

library_unit_body = ( subprogram_body | % package_body )

context_clause = (
  "" | % ( with_clause ( "" | % use_clause more_use ) context_clause ) )

/*******  more_use added for Gen *****/

more_use = ( "" | % 80 use_clause more_use )

with_clause = ( "with_clause " )

body_stub = ( "procedure_body_stub " | % "function_body_stub " | %
  "package_body_stub " | % "task_body_stub " )

subunit = ( "procedure_subunit " | % "function_subunit "
  | % "package_subunit " | % "task_subunit " )

exception_declaration = ( "exception_decl " )

exception_handler = ( "exception_handler " exception_choice
  sequence_of_statements )
```

```
exception_choice = ( "exception_choice_others " | % "predef_except " )

raise_statement = ( "raise_statement " )

generic_declaration = ( generic_specification )

generic_specification = ( generic_formal_part ( "gen_package_spec " | %
  "gen_subprog_spec " | % "gen_subprog_spec:function " | %
  "gen_subprog_spec:procedure " ) )

generic_formal_part = ( generic_parameter_declaration )

generic_parameter_declaration = ( "gen_formal_obj:default " | %
  "gen_formal_obj:in " | % "gen_formal_obj:in_default " | %
  "gen_formal_obj:in_out " | % "gen_formal_obj:no_default " | %
  "gen_formal_part " | % "gen_formal_subprog " | %
  "gen_formal_subprog:box_default " | % "gen_formal_subprog:nm_default " | %
  "gen_formal_type " | % "gen_formal_type:access " | %
  "gen_formal_type:array " | % "gen_formal_type:discrete " | %
  "gen_formal_type:fixed_point " | % "gen_formal_type:floating_point " | %
  "gen_formal_type:integer "  | % "gen_formal_type:lim_private " | %
  "gen_formal_type:private " )

generic_type_definition = ( "generic_type_definition " )

generic_instantiation = ( ( "gen_function_instantiation " | %
  "gen_package_instantiation " | % "gen_procedure_instantiation " | %
  "gen_subprog_instantiation " )  ( "" | % generic_actual_part ) )

generic_actual_part = ( "gen_actual_object " | % "gen_actual:subprog " | %
  "gen_actual:type " | % "gen_actual:type_access " | %
  "gen_actual:type_array " | % "gen_actual:type_discrete " | %
  "gen_actual:type_fixed_point " | % "gen_actual:type_floating_point " | %
  "gen_actual:type_integer " )

generic_association = ( "generic_association " )

generic_formal_parameter = ( "generic_formal_parameter " )

generic_actual_parameter = ( "generic_actual_parameter " )

representation_clause = (
  type_representation_clause | % address_clause )

type_representation_clause = ( length_clause
  | % enumeration_representation_clause
  | % record_representation_clause )

length_clause = ( "length_clause " | % "length_clause:size " | %
  "length_clause:small " | % "length_clause:strng_size " | %
  "length_clause:strg_size_access " | % "length_clause:strg_size_access " | %
  "length_clause:strg_size_task " )

enumeration_representation_clause = ( "enum_repr_clause " )

record_representation_clause = ( "record_repr_clause "  ( "" | %
  alignment_clause ) ( "" | % component_clause ) )

alignment_clause = ( "alignment_clause " )

component_clause = ( "component_clause " )

address_clause = ( "address_clause " )

code_statement = ( "code_statement " )
```

Appendix F. *ALIANT Operating Instructions and Output*

This appendix contains the operating instructions for the ALIANT prototype and samples of screen/file output. The first two sections contain the step by step interactive and batch operating instructions. The third sectio contains a complete output sample from an interactive ALIANT "session". Finally, the fourth section contains sample listings for the ALIANT support files not provided elsewhere.

F.1 Interactive Operating Instructions

The ALIANT prototype may be executed in interactive mode or batch mode. The interactive mode will be explained first. To execute ALIANT in interactive mode, enter the following statement:

```
runa* <grammar filename> <number of combinations>
```

The statement above includes two parameters. The first parameter is the filename of the input grammar without the ".gen" extension. The extension will be automatically appended to the first parameter; therefore the actual filename used in this example is "<grammar filename>.gen". The input grammar must be in Gen compatible format. The second parameter is the requested number of combinations to generate. This number must be greater than zero to be accepted by the ALIANT prototype. If the input grammar filename does exist and the number of requested combinations is greater than zero, ALIANT will begin execution.

The first stage of the prototype is the generation of the feature combinations by Gen. The following message is displayed while Gen is executing:

```
-------------------------------
-- Gen execution in progress --
-------------------------------
```

When Gen is finished, the ALIANT combination processing begins. For every tenth combination processed by the ALIANT Ada code, a dot is displayed on the screen as shown in the following example:

```
 ***************************************
 ** ALIANT initialization in progress **
 ***************************************

<newpage>

 *************************************************
 ** Processing Gen combinations, please wait... **
 *************************************************
....................
```

When all combinations have been processed, the ALIANT summary statistics are displayed, an audible tone sounds, and the user is prompted for input as shown below:

```
 ***** ALIANT Processing Statistics *****
 Number of combinations processed:  200
 Null combination count:  0
 Duplicate combination count:  78
 Resulting combinations:  122
 ****************************************


 Enter 'y' to SELECT combinations for display >>
```

The statistics show how many combinations were processed. The total number processed is further subdivided into the number that were null, duplicate or resulting combinations. The audible tone signals the completion of processing, which is helpful if a long interactive session is running "unattended". At this point, the user must decide if he/she wants to select combinations for display. If no selection is desired, a carriage return or any input not starting with an upper or lower case Y will pass control to the "load database" prompt. However, entering an upper or lower case Y will cause the following prompt to be displayed:

```
Enter duplication threshold >>
```

To select combinations for display, two threshold values must be entered. As indicated by the display above, the first threshold is the *duplication threshold.* Combinations will be selected that have duplication counts greater than or equal to the specified duplication threshold. The number input must be a natural number. After entering a valid threshold value, a similar prompt is displayed for the *feature threshold.* Combinations will be selected that contain no more features than the specified feature threshold. This number must also be a natural number.

```
Enter feature threshold >>
```

After both threshold values have been entered, the number of combinations that satisfy both limits is displayed followed by the paging prompt:

```
There are  9 combinations with a duplication count >=  4,
and a feature count <=  50.

Enter 'y' to DISPLAY selected combinations,
or 'p' to DISPLAY with paging >>
```

To display selected combinations continuously without paging, an upper or lower case Y is entered. If paging is desired, an upper or lower case P is entered. During the paging option, the user is given the opportunity to terminate paging with the following prompt:

```
Press RETURN to continue paging,
or enter 'q' to quit paging >>
```

If there were no combinations to display, the user chose not to display combinations, or combination display is complete, the following prompt is redisplayed:

```
Enter 'y' to SELECT combinations for display >>
```

The selection/display process can be repeated or terminated at this point by entering the appropriate option.

When the selection/display process is finally terminated, the following prompt is displayed for the database option:

```
Enter 'y' to load AFIS database >>
```

If this option is selected, the duplication and feature threshold values are entered as before using the following two prompts:

```
 Enter duplication threshold for database >>

<newpage>

 Enter feature threshold for database >>
```

After successful entry of these threshold values, the database load is started with the first message shown below, and completed with the second message shown below. The ALIANT session is then concluded with a normal termination message.

```
 Loading AFIS database...

 ...The AFIS database has been loaded with  6 records.

 ***************************
 ** Exiting ALIANT driver **
 ***************************
```

F.2 Batch Operating Instructions

To execute ALIANT in batch mode, an additional parameter is required at startup time:

```
runa* <grammar filename> <number of combinations> <batch filename> &
```

The batch filename is the complete filename of a text input file containing the entries that are normally input from the keyboard. A batch job can be executed in *background* by appending the ampersand as indicated above. A sample batch file is shown below:

```
y     (y = select combinations for display)
0     (0 = duplicate threshold)
10    (10 = feature threshold)
y     (y = display selected combinations)
n     (n = end combination selection)
n     (y = select combinations for database)
0     (0 = duplicate threshold)
10    (10 = feature threshold)
```

This batch file will display (continuously) all combinations with 10 or less features and load them into the AFIS database. The paging option should never be selected in a batch file since the batch file would also have to include the correct number of paging responses. The minimum batch file would simply allow the processing statistics to be recorded:

```
n     (n = don't select combinations for display)
n     (n = don't select combinations for database)
```

The output from an ALIANT batch execution is directed to the alnt_out file. This file includes the start and finish date/time as shown below for the minimum batch file example. Note that in batch mode, the responses input from the batch file do not show up in the alnt_out file.

```
Sat Sep  1 19:04:09 EDT 1990

<newpage>

 ***************************************
 ** ALIANT initialization in progress **
 ***************************************

<newpage>
```

```
 ************************************************
 ** Processing Gen combinations, please wait... **
 ************************************************
....................

 ***** ALIANT Processing Statistics *****
 Number of combinations processed:  200
 Null combination count:  0
 Duplicate combination count:  78
 Resulting combinations:  122
 ***************************************


 Enter 'y' to SELECT combinations for display >>

<newpage>

 Enter 'y' to load AFIS database >>

 ***************************
 ** Exiting ALIANT driver **
 ***************************

Sat Sep  1 19:04:30 EDT 1990
```

F.3 Sample Interactive Output

This section includes a complete example of an ALIANT interactive session. If the same user responses were provided in a batch input file, an ALIANT batch session would produce this same output in the alnt_out file. For clarity, all negative responses use the letter "n". In actual use, a simple carriage return or other entry will achieve the same purpose.

```
-------------------------------
-- Gen execution in progress --
-------------------------------
```

```
<newpage>

 **************************************
 ** ALIANT initialization in progress **
 **************************************

<newpage>

 **********************************************
 ** Processing Gen combinations, please wait... **
 **********************************************
...................

 ***** ALIANT Processing Statistics *****
 Number of combinations processed:  200
 Null combination count:  0
 Duplicate combination count:  78
 Resulting combinations:  122
 ***************************************


 Enter 'y' to SELECT combinations for display >> y

<newpage>

 Enter duplication threshold >> 4

<newpage>

 Enter feature threshold >> 50

<newpage>

 There are  9 combinations with a duplication count >=  4,
 and a feature count <=  50.

 Enter 'y' to DISPLAY selected combinations,
 or 'p' to DISPLAY with paging >> n

 Enter 'y' to SELECT combinations for display >> y

<newpage>

 Enter duplication threshold >> 3
```

```
<newpage>

 Enter feature threshold >> 30

<newpage>

 There are  11 combinations with a duplication count >=  3,
 and a feature count <=  30.

 Enter 'y' to DISPLAY selected combinations,
 or 'p' to DISPLAY with paging >> n

 Enter 'y' to SELECT combinations for display >> y

<newpage>

 Enter duplication threshold >> 5

<newpage>

 Enter feature threshold >> 10

<newpage>

 There are  6 combinations with a duplication count >=  5,
 and a feature count <=  10.

 Enter 'y' to DISPLAY selected combinations,
 or 'p' to DISPLAY with paging >> p

<newpage>

 Combination  4:
 ( 9 duplicate[s] )

 function_subunit                         (count: 1)

 Combination  9:
 ( 6 duplicate[s] )

 package_spec                             (count: 1)
 use_clause                               (count: 1)
 with_clause                              (count: 1)
 Combination  25:
 ( 5 duplicate[s] )

 use_clause                                  (count: 2)
 with_clause                                 (count: 1)
 function_subunit                            (count: 1)

 Combination  29:

 Press RETURN to continue paging,
 or enter 'q' to quit paging >>

 ( 7 duplicate[s] )

 procedure_subunit                           (count: 1)

 Combination  40:
 ( 6 duplicate[s] )

 subprogram_decl:procedure                   (count: 1)

 Combination  44:
 ( 6 duplicate[s] )

 use_clause                                  (count: 1)
 with_clause                                 (count: 2)
 procedure_subunit                           (count: 1)

 Enter 'y' to SELECT combinations for display >> n

<newpage>

 Enter 'y' to load AFIS database >> y

<newpage>

 Enter duplication threshold for database >> 5

<newpage>

 Enter feature threshold for database >> 10

<newpage>

 Loading AFIS database...
```

```
...The AFIS database has been loaded with  6 records.

**************************
** Exiting ALIANT driver **
**************************
```

F.4 Sample Support File Formats

This section contains sample listings for the following ALIANT support files not already displayed: gen_out, g_temp, and afis_db.

gen_out : Contains the output combinations from the Gen software.

```
START_COMPILATION: subprogram_decl:procedure subprog_param_spec:default
:END_COMPILATION

START_COMPILATION: procedure_body gen_package_instantiation gen_actual:type
null_statement if_statement if_statement label case_statement
simple_expression not_operator based_literal multiplication real_literal
exponentiation based_literal and_operator simple_expression unary_addition
based_literal exponentiation string_literal membership_test_in explicit_range
null_statement block_statement label assignment_statement simple_expression
unary_minus aggregate exponentiation based_literal rem_operator absolute_value
based_literal exception_handler exception_choice_others if_statement
null_statement if_statement label null_statement :END_COMPILATION

START_COMPILATION: with_clauce use_clause package_body null_statement label
label null_statement :END_COMPILATION

START_COMPILATION: function_subunit .END_COMPILATION

START_COMPILATION: with_clause function_subunit :END_COMPILATION

START_COMPILATION: with_clause use_clause with_clause use_clause
subprogram_decl:procedure :END_COMPILATION

START_COMPILATION: package_spec object_decl :END_COMPILATION

START_COMPILATION: with_clause use_clause with_clause use_clause package_body
null_statement assignment_statement simple_expression unary_addition
not_operator null_access_value division absolute_value null_access_value
catenation based_literal exponentiation null_access_value multiplication
not_operator real_literal membership_test_in explicit_range orelse
```

```
simple_expression null_access_value exponentiation based_literal
catenation null_access_value membership_test_in type_mark exception_handler
exception_choice_others label null_statement null_statement :END_COMPILATION

START_COMPILATION: package_spec object_decl :END_COMPILATION

START_COMPILATION: with_clause use_clause package_spec :END_COMPILATION
```

g_temp (as input to Gen): When used as input to Gen, this file contains the input grammar with the generation statement appended (i.e., "* 100 compilation").

```
graphic_character = ( "graphic_character " )

basic_graphic_character = ( "basic_graphic_character " )

basic_character = ( "basic_character " )

identifier = ( "identifier " )
.
. (majority removed for brevity, see Appendix E for complete grammar)
.

alignment_clause ) ( "" | % component_clause ) )

alignment_clause = ( "alignment_clause " )

component_clause = ( "component_clause " )

address_clause = ( "address_clause " )

code_statement = ( "code_statement " )

* 100 compilation
```

g_temp (as input to ALIANT_Driver): When used as input to the ALIANT_Driver, this only contains the number of combinations originally input as parameter 2. The previous contents of g_temp, the grammar and generation statement, are no longer needed and are overwritten by a single integer value.

```
1000
```

afis_db : Contains the bit matrix for the selected combinations (for brevity, only two records are shown).

```
>>>> AFIS DATABASE FILE <<<<

 4:  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

 9:  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 1 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

Bibliography

1. *Ada Compiler Validation Proccdures*. Technical Report AD-A210406, Washington D.C.: Ada Joint Program Office, May 1989.
2. *Ada Compiler Validation Summary Report: Verdix Corporation, VADS VAX UNIX, Version 5.5, DEC VAX 11/750*. Technical Report AD-A211623, Wright-Patterson AFB, OH: Ada Validation Facility, 1989.
3. Ada Validation Facility, Aeronautical Systems Division, Air Force Systems Command. DBMS-ALIANT-PAT Statement of Work - Task Order 17 Wright-Patterson AFB OH, 1988.
4. Aho, A. V. and J. D. Ullman. *Principles of Compiler Design*. MA: Addison-Wesley Publishing Company, April 1979.
5. Air Force Armament Laboratory. *Ada 9X Project Report/Plan*. Technical Report. Washington: Office of the Under Secretary of Defense for Acquisition, January 1989.
6. Anderson, Chris. "Ada 9X Project." Report to the Public, November 1989.
7. Bass, B. "DOD Issues Revised Ada Compiler Validation Tests," *Government Compvter News*, 7(18):53 (August 1988).
8. Bazzichi, F. and I. Spadafora. "An Automatic Generator for Compiler Testing," *IEEE Transactions on Software Engineering*, *SE-8*(4):343–353 (July 1982).
9. Berning, Paul T. and others. *Automated Compiler Test Case Generation*. Technical Report RADC-TR-78-30, Griffis AFB, NY: Naval Air Development Center, 1978.
10. Bertolino, A. and M. Fusani. "Software Validation: A Government-Imposed Challenge to the State of the Art in Certification," *Computer Standards and Interfaces*, *6*(4):433–436 (1987).
11. Burgess, C. J. "Towards the Automatic Generation of Executable Programs to Test a Pascal Compiler." In Barnes, D. and P. Brown, editors, *Software Engineering 1986*, pages 304–316, London: Peter Perigrinus Ltd., 1986.
12. Carlson, W. E. "Ada: A Promising Beginning," *Computer*, *14*(6):13–15 (June 1981).
13. Craine, D. B. *Ada Compiler Evaluation Techniques for Real-Time Avionics Applications*. MS thesis, Air Force Institute of Technology (AFIT), 1987.
14. DeMillo, R. and others. *Software Testing and Evaluation*. Menlo Park, California: Benjamin/Cummings, 1987.
15. Department of Commerce. *Software Validation, Verification, and Testing Technique and Tool Reference Guide*. NBS Special Publication 500-93. Washington D.C.: Government Printing Office, September 1982.
16 Department of Defense. *Reference Manual for the Ada Programming Language*. ANSI/MIL-STD 1815A. Washington D.C.: Government Printing Office, January 1983.
17. Drossopoulou, J. Uhl S. and others. *An Attribute Grammar for the Semantic Analysis of Ada*. Berlin: Springer-Verlag, 1982.
18. Eilers, D. E-Mail Correspondence. Irvine Compiler Corp., 11 November 1989.
19. Ganapathi, M. and G. O. Mendal. "Issues in Ada Compiler Technology," *Computer*, *22*(2):52–60 (February 1989).

20. Goepper, Eric R. *A Source Code Analyzer to Predict Compilation Time for Avionics Software Using Software Science Measures.* MS thesis, Air Force Institute of Technology (AFIT), 1988.

21. Goodenough, J. B. "The Ada Compiler Validation Capability," *Computer*, *14*(6):57–64 (June 1981).

22. – – – – –. *Ada Compiler Validation Implementers' Guide.* Technical Report, SofTech, Inc., 1986.

23. – – – – –. "Ada Compiler Validation: An Example of Software Testing Theory and Practice." In A., Habermann and U. Montanari, editors, *System Development and Ada - Proceedings of the CRAI Workshop on Software Factories and Ada*, pages 195–232, Berlin, W. G.: Springer-Verlag, 1987.

24. Herr, C. and others. "Compiler Validation and Reusable Ada Parts for Real-Time, Embedded Applications," *Ada Letters*, *8*(5):75–86 (Sept/Oct 1988).

25. Homer, W. and R. Schooler. "Independent Testing of Compiler Phases Using a Test Case Generator," *Software Practice and Experience*, *19*(1):53–62 (January 1989).

26. Johnson, S. C. *YACC - Yet Another Compiler Compiler.* Technical Report CSTR 32, Murray Hill, N.J.: Bell Laboratories, 1975.

27. Joyce, D. O. *Validating and Evaluating Ada's Representation Clauses and Implementation-Dependent Features.* MS thesis, Air Force Institute of Technology (AFIT), 1987.

28. Kasten, G. "A Test Case Generation Program." Software description, 18 June 1986.

29. Lee, P. *Realistic Compiler Generation.* MA: MIT Press, 1989.

30. Lesk, M. E. *Lex - A Lexical Analyzer Generator.* Technical Report CSTR 37, Murray Hill, N.J.: Bell Laboratories, 1975.

31. Mandl, R. "Orthogonal Latin Squares: An Application of Experiment Design to Compiler Testing," *Communications of the ACM*, *28*(10):1054–1058 (October 1985).

32. Oliver, P. "Experiences in Building and Using Compiler Validation Systems." In Merwin, R. and J. Zanca, editors, *AFIPS Conference Proceedings*, pages 1051–1057, New Jersey: AFIPS Press, June 1979.

33. Paprotney, George B. *LL - A Generator of Recursive Descent Parsers for LL(k) Languages.* MS thesis, Air Force Institute of Technology (AFIT), 1983.

34. Rennels, D. E-Mail Correspondence. New York University, NY., 14 September 1989.

35. – – – – – – and others. "Tool to Identify Ada Language Constructs in Source Code." Paper presented at the First Annual Armed Forces Communications and Electronics Association Mid-West Regional Conference. Dayton, Ohio, 18 July 1990.

36. Schmidt, David A. *Denotational Semantics - A Methodology for Language Development.* Boston: Allyn and Bacon, Inc., 1986.

37. Taylor, Tim T. Personal Correspondence. McDonnell Douglas Corporation, St. Louis MO, 2 January 1990.

38. *UNIX User's Manual - Reference Guide.* Berkeley, CA: 4.3 Berkeley Software Distribution, April 1986.

39. Wallace, Robert H. *Practitioner's Guide to Ada.* NY: McGraw-Hill, Inc., 1986.

40. Weiderman, N. H. *Ada Adoption Handbook: Compiler Evaluation and Selection.* Technical Report AD-A207717, Pittsburgh: Software Engineering Institute, 1989.

41. Wichmann, B. A. *Insecurities in the Ada Programming Language.* Technical Report DITC 137/89, United Kingdom: National Physical Laboratory, 1989.

42. ----- and M. Davies. *Experience with a Compiler Testing Tool.* Technical Report DITC 138/89, United Kingdom: National Physical Laboratory, 1989.

43. Williams, R. J. *Automatic Generation of Parsers Using Yacc and Lex.* MS thesis, Wright State University, OH, 1986.

44. Wilson, Steven P. Technical Director Ada Validation Facility. "Ada Features Identification System." Briefing to the AVF Managers Meeting, 8 June 1989.

REPORT DOCUMENTATION PAGE	*Form Approved* *OMB No. 0704-0188*

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE December 1990	3. REPORT TYPE AND DATES COVERED Master's Thesis
4. TITLE AND SUBTITLE AUTOMATIC DETERMINATION OF RECOMMENDED TEST COMBINATIONS FOR ADA COMPILERS		5. FUNDING NUMBERS
6. AUTHOR(S) James S. Marr, Captain, USAF		
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Air Force Institute of Technology,WPAFB OH 45433-6583		8. PERFORMING ORGANIZATION REPORT NUMBER AFIT/GCS/ENG/90D-09
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)		10. SPONSORING/MONITORING AGENCY REPORT NUMBER
11. SUPPLEMENTARY NOTES		
12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution unlimited		12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

Ada compilers are validated using the Ada Compiler Validation Capability (ACVC) test suite, containing over 4000 individual test programs. Each test program focuses, to the extent possible, on a single language feature. Despite the advantages of this "atomic testing" methodology, it is often the unexpected interactions between language features that result in compilation problems. This research investigated techniques to automatically identify recommended combinations of Ada language features for compiler testing. A prototype program was developed to analyze the Ada language grammar specification and generate a list of recommended combinations of features to be tested. While the skill and intuition of the compiler tester are essential to the annotation of the Ada grammar, the prototype demonstrated that automated support tools can be used to identify recommended combinations for Ada compiler testing.

14. SUBJECT TERMS Ada Compiler Validation, Compiler Testing, Grammars, Compiler Testing, Test Case Generators, Compilers, Ada Programming Language			15. NUMBER OF PAGES 219
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT UL